PRAGMATIC PERL

34

12/2015

pragmaticperl.com

Pragmatic Perl 34

pragmaticperl.com

Выпуск 34. Декабрь 2015

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

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

Авторы статей: Владимир Леттиев, Андрей Шитов, Денис Федосеев

Обложка: Марко Иваныk

Корректор: Андрей Шитов

Выпускающий редактор: Вячеслав Тихановский

Ревизия: 2015-12-31 10:36

Оглавление

1 От редактора 1

2 Впечатления от воркшопа Saint Perl 2015 2

3 Взгляд на 2015 г. 12

4 Управление модулями и прекомпиляция в Perl 6 16

5 Perl 6-винегрет 32

6 Использование Rust из Perl . 47

7 Обзор CPAN за ноябрь 2015 г. 90

8 Интервью с Дмитрием Шаматриным 105

1 От редактора

Всех читателей поздравляем с наступающими праздниками и желаем интересных проектов в новом году!

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

Приятного чтения.

■ *Вячеслав Тихановский*

2 Впечатления от воркшопа Saint Perl 2015

*Не изменяя традиции 19 декабря 2015 г. прошёл седьмой ежегодный воркшоп **Saint Perl** для всех любителей и профессионалов мира Perl в Санкт-Петербурге.*

Для меня незабываемые впечатления о конференции начались ещё с момента прибытия в Санкт-Петербург: затянутое серыми облаками небо, моросящий дождь и +6°C. Необычно для середины декабря, да и последующие дни ставили температурные рекорды.

В этом году конференция проходила на площадке, предоставленной генеральным партнёром конференции — компанией DataArt. На сайте конференции зареги-

стрировались 53 человека, причём трое из них сделали это уже в день конференции. Не все сразу смогли найти вход в здание, поэтому за 10 минут до начала зал был ещё почти пустой, но вскоре он заполнился, подошли, возможно, не все, но близко к заявленному числу. Была даже видеосъёмка, но получилась ли она и где будут записи, пока неизвестно.

Открыл конференцию Сергей Романов, поприветствовав слушателей и рассказав о плане конференции и намечающихся мероприятиях.

Первый доклад назывался «Протокол HTTP/2. Зачем и как использовать в Perl», и это был мой доклад. Мне трудно сказать, как восприняли доклад слушатели, но 40 минут для меня пролетели незаметно. Большая часть доклада была посвящена

протоколу HTTP: истории протокола, недостаткам HTTP/1.1 и достоинствам HTTP/2. Затем проводился обзор реализаций для Perl: подробно рассмотрен `Net::Curl` и его работа с HTTP/2, а также `Protocol::HTTP2` — как возможная основа для реализаций клиентов и серверов с поддержкой HTTP/2.

После доклада был вопрос о судьбе WebSocket, и для многих стало откровением, что WebSocket не работает поверх HTTP/2 и никакой альтернативы для его замены HTTP/2 не предлагает. Разработка спецификации WebSocket поверх HTTP/2 заглохла и похоже никому не нужна.

Следующим докладчиком должен был стать Илья Чесноков, который готовил интерактивный доклад, но из-за технической накладки доклад пришлось перенести. Поэтому следующим выступал Михаил

Матвеев с докладом «Инструменты для работы с изображениями в Perl».

В докладе был сделан обзор самых популярных модулей Perl для работы с изображениями: ImageMagick, GraphicMagick и другие. Как обычно, вопрос выбора модуля возник в результате работы над практической задачей: написание своей собственной капчи. Особо в докладе был выделен модуль Imager, который в отличие от других библиотек не требует установленных C-библиотек с заголовками (что иногда доставляет проблемы при сборке на некоторых дистрибутивах), так же написан с использованием C/XS, но собирается без проблем, работает быстрее, хоть и не поддерживает большого спектра форматов. Были представлены бенчмарки, в которых также можно было сравнить различия в скорости работы модулей.

Далее был обеденный перерыв, а затем Илья, для которого нашли переходник для видео-разъёма, смог подключить свой ноутбук и показать доклад «Технологии асинхронного программирования». Презентация была сделана с помощью Vroom — инструмента, позволяющего показывать слайды в редакторе vim. Доклад шёл целый час и фактически объединил в себе несколько больших докладов по различным техникам асинхронного программирования: это форки, треды и сопрограммы. Было рассказано, чем ужасны Perl треды и почему Coro — это единственные нормальная реализация тредов (хотя это и не совсем треды в привычном понимании). Бенчмарки иллюстрировали различия в работе различных реализаций асинхронных техник, как по нагрузке процессора, так и по использованию памяти.

Следующим докладчиком был Денис Федосеев, который представил доклад «Легаси это не страшно». Денис поделился опытом борьбы с огромным легаси-проектом, где нет ни тестов, ни разделения на пакеты, ни даже элементарного `use strict` (на этом месте особо чувствительные перловики должны упасть в обморок). Рассказал о техниках изоляции кода, способах ведения исправлений. Презентация была очень красочной, позволяя целиком проникнуться атмосферой безысходности: живые мертвецы, велосипеды с квадратными колёсами. Но в итоге был сделан вывод, что выживать с легаси можно, и это не страшно.

После кофе-брейка выступил лидер Moscow.pm Павел Щербинин с докладом «Index Condition Pushdown», который был посвящён одноимённой фиче, которая появилась в MySQL 5.6. Слайдов как

таковых было не много, в основном все объяснения зарисовывались на флипчарте, при этом очень активно происходило взаимодействие со слушателями, собственно это и было ближе к духу воркшопа (семинара), когда можно не только слушать, но и пытаться понять, о чём речь и пытаться отвечать на вопросы ;-)

Доклад закономерно породил вопрос: зачем MySQL, если есть PostgreSQL, но флейм был беспощадно отсечён — каждый использует то, что его устраивает и в чём он лучше разбирается.

В завершении конференции были блиц-доклады. Надо сказать, что большинство докладов были зарегистрированы уже по ходу конференции.

- Artur Penttinen сделал сразу два блиц-

доклада о том как можно распарсить конфигурационный файл однострочником и частный случай с конфигурационным файлов в формате ini.

- Илья Чесноков, «О регулярности». Немного философский доклад о событиях в нашей жизни и формировании полезных привычек, например, регулярное проведение конференций или ежегодное ведение цикла рождественских статей Perl Advent Calendar.

- Илья Чернов, «PSPlot — графики в EPS». Доклад был посвящён собственной разработке: модулю psplot, который позволяет рисовать двумерные графики в формате EPS.

- Доклады Михаила Иванова «SWAT» и Алексея Мележика «SparrowHub»

были логически объединены и посвящены системе Swat — DSL-языку и утилитам для быстрой разработки автоматических тестов web приложений и его спутнику sparrowhub, служащего для хранения сценариев тестирования для наиболее популярных веб-приложений (типа wordpress).

- Сергей Романов в докладе «Christmas is coming» наконец упомянул то, что практически не звучало на конференции: Perl 6. На рождество нас всех ждёт стабильный релиз Perl 6, которого так ждали последние 15 лет.

После окончания докладов последовала традиционная after party в одном из баров северной столицы (а для некоторых и не в одном). Но это уже другая история, впечат-

ления о которой трудно выразить словами, но можно измерить в единицах объёма ;-)

■ *Владимир Леттиев*

3 Взгляд на 2015 г.

Краткий обзор заметных событий в мире Perl за прошедший год

1 февраля

На конференции FOSDEM в Брюсселе Ларри Уолл объявил, что официальная бета-версия Perl 6 появится 27 сентября (на его 61-летие), а релиз 6.0 ожидается на Рождество 2015-го.

Статья в журнале с заметками с этого выступления: Perl 6, или Get ready to party.

2 февраля

Владимир Леттиев открыл сайт perlnews.ru — новостной блог о всех заметных событи-

ях мира Perl на русском языке.

16 февраля

Разработчики Rakudo объявили, что прекращают поддержку виртуальной машины Parrot, чтобы не тратить силы на поддержание бекенда, который все равно не сможет обеспечить некоторые нужные возможности, которые потребуются в предстоящем релизе Perl 6.0 в 2015 году.

6 апреля

The Perl Foundation объявили о создании фонда поддержки развития ядра Perl 6 (Perl 6 Core Development Fund).

28 апреля

Microsoft выпустил текстовый редактор для

программистов Visual Studio Code. Среди языков, для которых доступна подстветка синтаксиса, — Perl и Perl 6.

1 июня

Вышел релиз Perl 5.22. Статьи в журнале:

- Что нового в Perl 5.22
- Перевод документа perl5220delta.pod на русский язык

6 июня

Марк Леман анонсировал свой форк Perl 5.22, провокационно назвав его stableperl.

16 августа

20 лет CPANу.

27 сентября

Perl 6 (в лице Rakudo Star 2015.09) официально получил статус бета-версии.

18 декабря

Языку программирования Perl исполнилось 28 лет.

25 декабря

Релиз Perl 6.

■ *Андрей Шитов*

4 Управление модулями и прекомпиляция в Perl 6

25 декабря 2015 г. вышел первый стабильный релиз Rakudo Perl 6, среди новшеств которого совершенно новая система управления модулями и прекомпиляция. Рассмотрим в деталях процесс загрузки, разрешения зависимостей и компиляции модулей.

Обычная загрузка модуля

Традиционный процесс загрузки модуля в Perl 5 выглядит достаточно просто: имя модуля преобразуется к имени файла, например, `Foo::Bar` ожидается быть найденным в файле Foo/Bar.pm. В массиве `@INC` содержится список каталогов, в которых следует производить поиск файлов модулей. Часть

каталогов является фиксированными и получили свои обозначения, например:

- *site* — каталог для поиска модулей, установленных администратором системы,
- *vendor* — каталог для модулей, поставляемых с дистрибутивом,
- *perl* — каталог модулей, идущих в поставке Perl

Модули ищутся и загружаются во время компиляции программы из исходного кода на диске. Такой же метод до недавнего времени использовался и в Perl 6, но рождественский релиз внёс существенные изменения в такое положение вещей.

Репозитории

Теперь в Perl 6 мы работаем с репозиториями — определённым образом организованное хранилище модулей. Репозитории могут быть различных типов, и это первое отличие от традиционных каталогов. Простейший вариант репозитория совместим с традиционным подходом и хранит файлы модулей в соответствии с именем пакета. Более сложный вариант поддерживает установку дистрибутивов модулей вместе с метаинформацией и ресурсами, а также производит прекомпиляцию модулей в байткод.

Ранее используемый массив `@*INC` ушёл в небытие и теперь при загрузке модулей используется переменная процесса `PROCESS::<$REPO>`, которая является связанным списком с информацией о

доступных репозиториях.

```
$ perl6 -e 'for $*REPO.repo-chain
    -> $n { say $n }'

inst#/home/user/.perl6/2015.12
inst#/usr/share/perl6/site
inst#/usr/share/perl6/vendor
inst#/usr/share/perl6
...
```

Как видно, в отличие от Perl 5, в цепочке репозиториев присутствует новый каталог, находящийся в каталоге пользователя, запустившего программу, и он имеет максимальный приоритет. Кроме того, присутствует префикс `inst#`, который указывает на тип репозитория, в данном случае тип `CompUnit::Repository::Installation`, который поддерживает инсталляцию.

По-прежнему мы можем дополнять список с помощью флага –I или переменной окружения PERL6LIB, например:

```
$ perl6 –I /opt/perl6 'for $*REPO
    .repo–chain –> $n { say $n }'

file#/opt/perl6
inst#/home/user/.perl6/2015.12
inst#/usr/share/perl6/site
inst#/usr/share/perl6/vendor
...
```

Как видно, если не указать тип репозитория перед путём, то он будет считаться традиционным файловым `file#`.

Что касается выражения `use lib '/path/to/lib'`, то теперь её нельзя использовать в модулях, только в скриптах.

Инсталляция в репозиторий

Инсталляция в традиционный файловый репозиторий — это простое копирование файлов. С новым типом репозитория всё несколько сложнее. Рассмотрим как организован такой репозиторий:

```
\
 +– dist
 |
 +– sources
 |
 +– resources
 |
 +– short
 |
 +– precomp
    \
     + compiler.id
      \
       + ..
```

При инсталляции дистрибутива `Foo`, содержащего модуль `Foo::Bar`, в каталоге dist создаётся файл с именем, представляющим собой SHA1 от уникального идентификатора дистрибутива, содержащий метаинформацию о дистрибутиве в формате JSON. В каталог sources помещаются исходные коды модуля, в каталог resource - связанные ресурсные файлы, все имена файлов генерируются как уникальные 40-символьные идентификаторы.

В каталог short помещаются файлы с именами как результат хеш-функции SHA1 от имени соответствующего модуля. В таком файле помещается 40-символьный идентификатор дистрибутива, в состав которого входит данный модуль.

В каталог precomp помещается результат компиляции исходного кода в байткод.

Внутри каталога precomp создаётся каталог для текущей версии компилятора (в имя включается даже время сборки компилятора). Дальнейшая организация вложенных каталогов precomp напоминает git: внутри создаётся до 256 двухбуквенных каталогов, в который помещаются скомпилированный байткод, например:

```
+— 5C
   \
    + 5
      C74B123E79B373BB96EC583A5C7EE4

    + 5
      C74B123E79B373BB96EC583A5C7EE4
      .deps
    + 5
      C74B123E79B373BB96EC583A5C7EE4
      .rev—deps
+— 7E
   \
    + 7
```

```
        E4AC40F59629C48CCBD390FEB4B27
9       + 7
          E4AC40F59629C48CCBD390FEB4B27
          .deps
10      + 7
          E4AC40F59629C48CCBD390FEB4B27
          .rev-deps
```

Создаются также файлы .deps и .rev-deps куда помещается данные о зависимостях и обратных зависимостях (sha1-хеш имени модуля).

Таким образом, когда в коде встречается запись `use Foo::Bar`, компилятор выполняет такую цепочку действий:

1. вычисляет sha1-сумму имени `Foo::Bar`;

2. ищет файл с получившимся id в каталоге short;
3. если находит, то по содержимому узнаёт id дистрибутива;
4. загружает метаинформацию дистрибутива из каталога dist;
5. из метаинформации узнаёт расположение исходного кода нужного модуля дистрибутива;
6. проверяет наличие исходного кода в каталога sources (но не загружает);
7. проверяет наличие прекомпилированного кода в precomp;
8. если нет прекомпилированного кода, то выполняет загрузку исходного кода и прекомпиляцию с сохранением байткода;
9. загружает из каталога precomp файл с прекомпилированным кодом модуля;
10. проверяет наличие в каталоге

precomp файла с таким же именем, но с окончанием .deps;

11. если есть deps-файл, то находит в нём имена source-файлов зависимостей;
12. в цикле проходит шаги 2–12 для всех зависимостей.

За инсталляцию отвечает класс `CompUnit::Repository::Installation`. Пример кода, выполняющего установку, можно увидеть в скрипте установки дистрибутива CORE при сборке Rakudo. В целом примерно так:

```
my %provides =
    "Foo::Bar" => "lib/Foo/Bar.pm6",
;

# задаём путь к репозиторию для данного процесса
```

```
PROCESS::<$REPO> := CompUnit::
    RepositoryRegistry
     .repository-for-spec("inst#/
        path/to/repo");

# устанавливаем/прекомпилируем
$*REPO.install(
    Distribution.new(
        name     => "Foo",
        auth     => "vendor",
        ver      => "0.1",
        provides => %provides,
    ),
    %provides,
    :force,
);
```

Недостатки

К сожалению, скорость, с которой появились новые репозитории (хотели успеть к Рождеству), помешала достаточно хорошо протестировать и нормально реализовать функционал.

`$(DESTDIR)`

Большинство дистрибутивов Linux и UNIX-систем используют переменную окружения `$(DESTDIR)`, чтобы указать временный каталог для инсталляции при сборке (например, при сборке пакетов). Новый механизм репозиториев затрудняет указание временного префикса, поэтому вышедший Rakudo 2015.12 просто падает с ошибкой при установке с использованием

`$(DESTDIR)`.

К счастью, это быстро заметили и ввели новую переменную окружения `RAKUDO_PREFIX`, которая позволяет переопределить префикс при операциях с репозиториями. Но это уже войдёт только в следующий релиз.

Прекомпиляция

Использование результатов прекомпиляции и сама прекомпиляция выполняется только в самом первом репозитории из списка. По умолчанию это репозиторий из домашнего каталога пользователя. Даже если исходный код находится, например, в репозитории *vendor* и там же есть прекомпилированный код, то rakudo не будет использовать результат прекомпиляции, а заново выполнит прекомпиляцию и

создаст необходимые файлы в каталоге пользователя.

В этом отношении каталоги прекомпиляции выглядят вообще отдельной сущностью от репозиториев.

Процесс прекомпиляция каждого модуля выполняется в отдельном процессе (exec), что безумно затормаживает первый запуск программы, импортирующей множество модулей.

Более того, если репозиторий окажется недоступным для записи пользователю, то программа просто не запустится с ошибкой.

Rakudo 2016.01

16 января 2016 г. ожидается очередной релиз Rakudo. Вероятно, что-то из обнаруженных проблем будет исправлено.

■ *Владимир Леттиев*

5 Perl 6-винегрет

Новогодний оливье — сборная солянка про разные интересные штуки в Perl 6

Юникод

Хранение и обработка символов внутри строк Perl 6 происходит в формате NFG (Normal Form Grapheme). Это название придумано еще в бытность Parrot, а с практической точки зрения достаточно знать, что из любого символа можно получить его NFC-, NFD-, NFKC- и NFKD-формы :-) Это разные канонические и декомпозированные представления символов в юникоде.

Подробности про разные формы представ-

лений с интересными примерами опубликованы на сайте юникода, а история NFG — в блоге Джонатана.

Посмотреть разные формы помогут одноименные методы, которые можно вызывать на односимвольных строках:

```
say $s.NFC; # codepoint
say $s.NFD;
say $s.NFKC;
say $s.NFKD;
```

Полное каноническое название возвращает метод `.uniname`:

```
say 'й'.uniname; # CYRILLIC SMALL
    LETTER SHORT I
```

У строк есть полезный метод `.encode`, позволяющий получить представление в той или иной юникодной кодировке:

```
my $name = 'Перл';
say $name.encode('UTF-8');

# utf8:0x<d0 9f d0 b5 d1 80 d0 bb
    >
```

(Этот метод не поможет перекодировать CP1215 в КОИ-8.)

Для закрепления материала предлагаю посмотреть на вывод упомянутых методов на разных хитрых символах:

```
unidump('□');
unidump('ы');
unidump('å');
unidump('é');
unidump('□'); # один из немногих
    символов,
                # для которого
                   различаются
                # все канонические
                   представления
```

```
unidump('й');
unidump('²');
unidump('Æ');

sub unidump($s) {
    say $s;

    say $s.chars; # число графем
    say $s.NFC;   # code point
    say $s.NFD;
    say $s.NFKC;
    say $s.NFKD;
    say $s.uniname; # юникодное
       название буквы

    say $s.NFD.list; # списком

    say $s.encode('UTF-8').elems;
        # сколько байтов
    say $s.encode('UTF-16').elems
       ;

    say $s.encode('UTF-8'); # в
```

```
        виде utf8:0x<...>
28
29      say '';
30  }
```

Формы NFKC и NFKD, в частности, могут преобразовать надстрочные и подстрочные индексы в обычные цифры:

```
1 say '2'.NFKD; # NFKD:0x<0032>
2 say '²'.NFKD; # NFKD:0x<0032>
```

С помощью функции `unimatch` можно узнать, принадлежит ли символ к той или иной группе символов в юникоде. Например:

```
1 say unimatch('ф', 'Cyrillic'); #
     True
```

Но нужно быть осторожным, чтобы не разжигать:

```
say unimatch('ї', 'Cyrillic'); # True
say unimatch('ї', 'Cyrillic'); # False
```

Здесь были одинаково выглядящие, но разные символы. Их NFD соответственно `0x<0456 0308>` и `0x<0069 0308>`, а имена — `CYRILLIC SMALL LETTER YI` и `LATIN SMALL LETTER I WITH DIAERESIS`.

Юникодные свойства можно проверить и регулярными выражениями:

```
say 1 if 'э' ~~ /<:Cyrillic>/;
say 1 if 'э' ~~ /<:Ll>/; # Letter lowercase
```

Для создания юникодных строк можно напрямую воспользоваться конструктором класса `Uni`:

```
say Uni.new(0x0439).Str;      # й
```

```
say Uni.new(0xcf, 0x94).Str; # Ï
```

Whatever (*)

Whatever — это класс, который позволяет создавать объекты, тип и значение которых определяются из контекста. Чтобы создать такой объект, достаточно поставить звездочку (если она, разумеется, не означает умножение).

```
say *.WHAT; # (Whatever)
```

Whatever удобно использовать для создания простых функций, например:

```
my $f = * ** 2; # возведение в
    квадрат
say $f(16);     # 256

```

```
my $xy = * ** *; # возведение в
    произвольную степень
say $xy(3, 4);   # 81
```

Того же результата удастся достичь более традиционным (для старого Perl 6) синтаксисом:

```
# Анонимный блок кода с одним
    аргументом $x
my $g = -> $x {$x ** 2};
say $g(16);

# Блок с двумя аргументами, имена
    сортируются по алфавиту
my $h = {$^a ** $^b};
say $h(3, 4);
```

Вполне оправдано использование звездочки для создания интервалов (Range):

Пример с циклом:

```
for (5..*) {
    .say; # построчно печатает от
        5 до 10
    last if $_ == 10;
}
```

Пример с массивом:

```
my @a = <2 4 6 8 10 12>;
say @a[3..*]; # (8 10 12)
```

Отдельно следует рассмотреть, что происходит с отрицательными значениями.

Все, начиная с четвертого элемента и заканчивая предпоследним:

```
say @a[3..*−2]; # (8 10)
```

Чтобы получить последний элемент, нельзя просто написать `@a[−1]`:

```
Unsupported use of a negative -1
    subscript to index from the
    end;
in Perl 6 please use a function
    such as *-1
```

Следует добавить звездочку:

```
say @a[*-1]; # 12
```

Звездочка подходит и для создания ленивых списков (то есть таких списков, очередной элемент которых вычисляется по мере необходимости):

```
my @a = (1 ... *);
for (0..5) {
    say @a[$_];
}
```

В ленивых списках допустимо подсказать компилятору, как генерировать последова-

тельность:

```
my @a = (1, 2 ... *);     # с шагом 1
my @a = (2, 4 ... *);     # четные числа
my @a = (2, 4, 8 ... *); # степени двойки
```

Либо использовать явно определенный блок со способом вычисления:

```
my @fib = 0, 1, * + * ... *; # Фибоначчи
```

Файлы

Имя файла, который сейчас исполняется, теперь находится в переменной `$*PROGRAM-NAME`, а не `$*PROGRAM_NAME`, как было еще совсем недавно.

А вот программа, которая использует встроенную функцию `slurp`, читающую содержимое текущего файла:

```
say slurp $*PROGRAM_NAME;
```

Противоположная функция — `spurt` («бить струей»), она записывает данные в файл. Вот пример реализации юниксовой функции `cp` на Perl 6:

```
my ($source, $dest) = @*ARGS;

my $data = slurp $source;
spurt $dest, $data;
```

По умолчанию файл либо создается, либо перезаписывается. Ключ (именованный аргумент) `:append` попросит добавлять новые данные в конец файла:

```
spurt $dest, $data, :append;
```

Чтобы упасть с ошибкой при попытке записи в существующий файл, добавьте опцию `:createonly` (`:append` при этом станет неактуальной):

```
spurt $dest, $data, :createonly;
```

И `slurp`, и `spurt` принимают аргумент с именем кодировки `enc`, но опять же, старых кодировок типа `'KOI8'` там нет (да и не надо):

```
my $data = slurp $source, enc => 'UTF-8';
spurt $dest, $data, enc => 'UTF-16';
```

Если на строке вызвать метод `.IO`, то возвращается объект класса IO::Path. Метод определен так:

```
method IO() returns IO::Path:D
```

(:D здесь не ржачный смайл, а означает defined.)

Например, как получить абсолютный путь к локальному файлу:

```
say 'file1.pl'.IO.abspath;
```

Как получить содержимое такого-то каталога:

```
say '.'.IO.dir;
```

Проверки существования файлов и каталогов теперь выглядят так:

```
say 1 if 'file.txt'.IO.e; # –e '
    file.txt' в Perl 5
say 1 if 'file.txt'.IO.f; # –f
say 1 if '..'.IO.d;       # –d
```

Остальные методы, перечисленные в документации, вполне понятны без объяс-

нений. Например, для выделения из имени файла расширения можно воспользоваться методом `.extension`:

```
say $filename.IO.extension;
```

■ *Андрей Шитов*

6 Использование Rust из Perl

Встраивание Rust в Perl с помощью FFI

`Rust` — язык системного программирования, развиваемый Mozilla Fundation. Призван заменить собой C. Обеспечивает безопасную работу с памятью и прочие приятные мелочи, которых иногда так не хватает в C, при этом не дает оверхеда по производительности (в теории).

Подробности языковых конструкций я тут расписывать сильно не буду, в целом, все примеры укладываются в первые 4 главы книги по Rust.

Сначала установим `Rust` последней версии (1.5 на момент написания). Инструкции по установке на все поддерживаемые

платформы можно найти на сайте — https://www.rust-lang.org/downloads.html.

Будем считать, что `Rust` уже установлен и даже работает. Попробуем его использовать из `Perl`.

Сначала опробуем пример из документации, создадим новый проект с помощью `Cargo`:

```
cargo new embed
cd embed
```

Добавим в `Cargo.toml` следующее содержимое:

```
[lib]
name = "embed"
crate-type = ["dylib"]
```

Этим мы говорим карго, что мы хотим

получить библиотеку, а не бинарник и библиотека должна линковаться с кодом на C.

Теперь отредактируем `src/lib.rs` и приведем его к такому виду:

```
use std::thread;

#[no_mangle]
pub extern fn process() {
    let handles: Vec<_> = (0..10)
        .map(|_| {
         thread::spawn(|| {
             let mut x = 0;
             for _ in (0..5
                _000_000) {
                 x += 1
             }
             x
         })
    }).collect();

```

```
15      for h in handles {
16          println!("Thread finished
                with count={}",
17          h.join().map_err(|_| "
               Could not join a
               thread!").unwrap());
18      }
19
20      println!("done!");
21  }
```

Это приложение запускает 10 тредов, каждый из которых считает до 5 000 000, собирает от них возвращаемое значение (счетчик `x`) и выводит это нам.

Соберем его:

```
1 cargo build --release
```

В каталоге `target/release` у нас появится файл `libembed.so` (для Mac OS расшире-

ние будет dylib, для Win — dll, я все это проделываю на маке, так что библиотеки везде будут с расширением dylib, вам же надо будет поправить под свою систему) — это наша подключаемая библиотека. При формировании библиотеки `Cargo` автоматически добавляет префикс `lib` к имени проекта, указанного в `name`.

Настало время вызвать ее из `Perl`. Так как XS-биндингов в `Perl` для `Rust` еще не существует (ну или я их не нашел), то мы будем использовать FFI::Raw. Этот модуль позволяет из кода на `Perl` дергать функции в библиотеках на C. Модуль весьма низкоуровневый, но для начала это даже хорошо.

Создадим файл `main.pl` со следующим содержимым:

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

```
2
3 use v5.16;
4 use FFI::Raw;
5
6 my $process = FFI::Raw->new(
7   'target/release/libembed.dylib'
        , 'process',
8   FFI::Raw::void,
9 );
10
11 say $process->call();
12
13 say "Done!";
```

Здесь в конструкторе мы указываем путь к нашей библиотеке, имя функции, которую мы хотим использовать, и тип возвращаемого значения. В данном случае функция у нас ничего не возвращает, так что это будет `void`.

И запустим наш перловый скрипт:

```
alpha6$ perl ./main.pl
Thread finished with count
   =5000000
Thread finished with count
   =5000000
Thread finished with count
   =5000000
Thread finished with count
   =5000000
Thread finished with count
   =5000000
Thread finished with count
   =5000000
Thread finished with count
   =5000000
Thread finished with count
   =5000000
Thread finished with count
   =5000000
Thread finished with count
   =5000000
done!
Done!
```

Этот код может валиться с `Segmentation fault 11` перед выходом из скрипта. Это связано с выводом с STDOUT из кода на `Rust`. Видимо, FFI как-то криво обрабатывает закрытие дескрипторов. На маке валится стабильно, на Linux работает без ошибок, на Windows не проверял. Видимо, влияют какие-то особенности платформы.

Вызывать числодробилки из `Perl` это конечно хорошо, но как насчет передачи параметров функции? Тоже можно, посчитаем факториал.

Приведем src/lib.rs к виду:

```
#[no_mangle]
pub extern fn fact(x: i32) -> i32 {
    let mut fact = 1;
    for i in 1..x+1 {
        fact *= i;
```

```
6       }
7
8       return fact;
9   }
```

Здесь мы объявляем публичную функцию `fact`, которая принимает на вход число в виде 32-битного целого и то же самое возвращает в качестве результата.

Так же обратите внимание: `#[no_mangle]` надо указывать перед каждой функцией, которую мы хотим экспортировать. Если убрать эту строку и попробовать вызвать рассчет факториала, мы получим ошибку, несмотря на то, что функция объявлена публичной:

```
1 alpha6$ perl ./main.pl
2 dlsym(0x7ffeda40abd0, fact):
      symbol not found at ./main.pl
      line 13.
```

no_mangle говорит компилятору, чтобы он не изменял имя функции во время компиляции, что делает возможным доступ к ней из других языков.

Приведем main.pl к виду:

```
my $fact_fn = FFI::Raw->new(
  'target/release/libembed.dylib'
     , 'fact',
  FFI::Raw::int,
  FFI::Raw::int,
);

my $fact_val = $fact_fn->call(30)
    ;

say "Factorial [$fact_val]";
```

Сохраняем, запускаем:

```
alpha6$ perl ./main.pl
Factorial [1409286144]
```

```
3 Perl code done!
```

Хм, кажется, что-то пошло не так...

В этом примере можно увидеть наглядно различия `Perl` и языков более низкого уровня. Если посмотреть на результат работы функции, то видно что факториал 30 немножко не является факториалом 30, хотя бы потому что в нем должно быть 22 цифры. При этом просто увеличение разрядности до u64 не поможет, потому что туда он тоже не влезает. В общем, мораль сей басни такова — следите за переполнением переменных, так как компилятор этого не отслеживает, и рантайм-проверок по умолчанию тоже нет.

Впрочем, при изменении разрядности с int до int64, факториал 14 рассчитывается верно, если верить калькулятору, а все, что вы-

ше 15 — это уже математика с хитрыми алгоритмами, `Rust` из коробки с ней работать не умеет.

Еще одна проблема — `FFI::Raw` не позволяет напрямую передавать и возвращать из внешнего кода что-то сложнее примитивных типов. То есть передать в функцию хеш или вернуть вектор не получится. Но всегда можно работать со структурами C, а перегнать Perl-структуру в сишную — задача элементарная.

Приведем наш `lib.rs` к виду:

```
1 pub struct Args {
2     init: u32,
3     by: u32,
4 }
5
6 #[no_mangle]
7 pub extern fn use_struct(args: &
```

```
    Args) -> u32 {
8     println!("Args: {} - {}",
          args.init, args.by);
9
10    return args.init;
11 }
```

Здесь мы объявляем структуру Args и делаем ее доступной снаружи нашей библиотеки указанием pub. В функции use_struct мы говорим компилятору, что ожидаем на входе указатель на эту структуру. Ну и возвращаем обратно первый элемент структуры.

В перловый код вставим такое:

```
1 my $packed = pack('LL', 42, 21);
2 my $arg = FFI::Raw::MemPtr->
    new_from_buf($packed, length
    $packed);
3
```

```
my $foo = FFI::Raw->new(
  $shared, 'test_struct',
  FFI::Raw::uint,
  FFI::Raw::ptr,
);

my $val = $foo->($arg); #Получаем структуру из Rust кода
say "result [$val]";
```

Здесь мы упаковываем данные для структуры (два числа int32), затем создаем из них объект `FFI::Raw::MemPtr`, который выделяет область памяти и укладывает туда упакованную стуктуру. Затем мы привязываемся к функции `test_struct` из нашей библиотеки и говорим, что будем передавать в нее указатель.

Соберем и запустим:

```
alpha6$ cargo build --release
```

```
alpha6$ perl ./main.pl
Args: 42 — 21
result [42]
```

Поздравляю, мы только что передали структуру данных в нашу библиотеку на `Rust`. Попробуем теперь сделать с этой структурой что-то более осмысленное.

Создадим счетчик (действие не особо осмысленное, зато показывает как с этим всем работать). Приведем наш `lib.rs` к виду:

```
#![allow(non_snake_case)]
use std::mem::transmute;

pub struct Args {
    init: u32,
    by: u32,
}

```

```
pub struct Counter {
    val: u32,
    by: u32,
}

impl Counter {
    pub fn new(args: &Args) -> Counter {
        Counter{
            val: args.init,
            by: args.by,
        }
    }

    pub fn get(&self) -> u32 {
        self.val
    }

    pub fn incr(&mut self) -> u32 {
        self.val += self.by;
        self.val
    }
```

```
30
31      pub fn decr(&mut self) -> u32
            {
32          self.val -= self.by;
33          self.val
34      }
35  }
36
37  #[no_mangle]
38  pub extern fn createCounter(args:
        &Args) -> *mut Counter {
39      let _counter = unsafe {
            transmute(Box::new(Counter
            ::new(args))) };
40      _counter
41  }
42
43  #[no_mangle]
44  pub extern fn getCounterValue(ptr
        : *mut Counter) -> u32 {
45      let mut _counter = unsafe { &
            mut *ptr };
46      let val = _counter.get();
```

```
47      return val;
48  }
49
50  #[no_mangle]
51  pub extern fn incrementCounterBy(
        ptr: *mut Counter) -> u32 {
52      let mut _counter = unsafe { &
            mut *ptr };
53      _counter.incr()
54  }
55
56  #[no_mangle]
57  pub extern fn decrementCounterBy(
        ptr: *mut Counter) -> u32 {
58      let mut _counter = unsafe { &
            mut *ptr };
59      _counter.decr()
60  }
61
62  #[no_mangle]
63  pub extern fn destroyCounter(ptr:
        *mut Counter) {
64      let _counter: Box<Counter> =
```

```
        unsafe{ transmute(ptr) };
    // Drop
}
```

Этот код я честно утащил отсюда, правда он не заработал на новой версии компилятора, да и с `FFI::Raw` у него тоже любовь не сложилась, пришлось его немного поправить. Но ничего, нам, пользователям `Mojolicious`, не привыкать.

Что здесь происходит? Сначала создаются две структуры данных: `Args` и `Counter`. В целом они одинаковые, но для `Counter` мы создаем реализацию, которая добавляет набор методов к нашей структуре.

Затем мы объявляем публичную функцию `createCounter`, которая на вход принимает указатель на структуру данных `Args`. Из этой структуры мы создаем объект

`Counter`, который упаковываем в контейнер `Box::new`, который будет доступен через указатель. Работа с `transmute` дело опасное, поэтому мы оборачиваем создание контейнера в `unsafe` и этим самым мы говорим компилятору, что мы сами позаботимся о проверке на безопасность всей этой конструкции (чего мы не делаем в данном случае).

После создания контейнера мы возвращаем из функции изменяемый указатель, с которым и будем дальше работать везде.

Теперь создадим файл `counter.pl`, который будет использовать нашу новую библиотеку:

```
1 #!/usr/bin/env perl
2
3 use v5.16;
4 use FFI::Raw;
```

```
5
6  my $shared = 'target/release/
       libembed.dylib';
7
8  my $packed = pack('LL', 42, 21);
9  my $arg = FFI::Raw::MemPtr->
       new_from_buf($packed, length
       $packed);
10
11 my $createCounter = FFI::Raw->new
       (
12   $shared, 'createCounter',
13   FFI::Raw::ptr, FFI::Raw::ptr
14 );
15
16 my $getCounterValue = FFI::Raw->
       new(
17   $shared, 'getCounterValue',
18   FFI::Raw::uint, FFI::Raw::ptr
19 );
20
21 my $incrementCounterBy = FFI::Raw
       ->new(
```

```
22   $shared, 'incrementCounterBy',
23   FFI::Raw::uint, FFI::Raw::ptr
24 );
25
26 my $ptr = $createCounter->($arg);
27
28 my $val = $getCounterValue->($ptr
       );
29 say "Current value [$val]";
30
31 my $nval = $incrementCounterBy->(
       $ptr);
32 say "New value [$nval]";
```

Здесь мы сначала привязываемся к функции `createCounter` и говорим, что будем передавать указатель и ждем на выходе указатель. Затем привязываемся к функции `getCounterValue` и говорим, что передаем ей указатель и ждем unsigned int. Для всех остальных функций должны быть созданы аналогичные записи, здесь я их

описывать не стал.

Дальше мы вызываем `createCounter`, которому передаем собранную руками структуру `Args` и получаем обратно указатель на контейнер со счетчиком. Дальше мы используем этот указатель для изменения значений счетчика.

Соберем это все дело и запустим:

```
alpha6$ cargo build --release
alpha6$ perl ./main.pl
Current value [42]
New value [63]
```

В выводе видно, что мы получили текущее значение счетчика после создания, а потом изменили его значение.

Теперь мы умеем получать структру из `Rust` и передавать ее обратно в библиоте-

ку. Однако есть проблема — если мы хотим использовать эту структуру внутри `Perl` кода, то нас ждет небольшое разочарование. У меня так и не получилось привести то, что возвращает `Rust`, во что-то пригодное для использования в `Perl`. Если кто-то знает, как это сделать, — напишите в комментариях, а?

Но структуры-то получать хочется! Что ж, пришло время добавить еще один слой абстракции — используем FFI::Platypus вместо `FFI::Raw`.

`FFI::Platypus` является намного более суровым инструментом и предоставляет кучу опций, позволяя не заморачиваться с низкоуровневой работой с данными в отличие от `FFI::Raw`.

Проверим, что у нас все работает как

надо. Установим FFI::Platypus и FFI::Platypus::Lang::Rust, этот модуль позволяет использовать при объявлениях структур и функций структуры из Rust, а не запоминать маппинг между Rust и C.

Изменим lib.rs и создадим файл plat.pl с содержимым, указанным ниже.

src/lib.rs:

```
#[no_mangle]
pub extern fn hello_plat(arg: u32) {
    println!("Hello PL {}", arg);
}
```

plat.pl:

```
#!/usr/bin/env perl
use v5.16;
use FFI::Platypus;

```

```
5 my $shared = 'target/release/
      libembed.dylib';
6
7 my $ffi = FFI::Platypus->new();
8
9 $ffi->lib($shared);
10 $ffi->lang('Rust');
11 $ffi->attach( hello_plat => ['u32
      '] => 'void' );
12 hello_plat(1);
```

Здесь мы создаем объект `FFI::Platypus` и указываем ему путь к библиотеке — `$ffi->lib($shared)`, затем указываем, какой язык мы хотим использовать `$ffi ->lang('Rust')`. После этого указания FFI сам подгрузит модуль `FFI::Paltypus ::Lang::<LangName>`, и можно будет использовать фичи для конкретного языка (соответствующий модуль должен быть предварительно установлен).

Далее мы добавляем функцию из библиотеки в перловый код, конструкция `$ffi ->attach` создает функцию с указанным именем, которая транслирует данные в/из соответстующую библиотечную функцию. Функция создается в пространстве имен модуля, так что надо следить, чтобы не было перекрытия функций из библиотеки и текущего пакета.

Собираем, запускаем:

```
alpha6$ cargo build --release
alpha6$ perl ./plat.pl
Hello PL 1
```

Все работает как и задумано.

Настало время получить желанную структуру из нашей библиотеки.

В очередной раз переделаем наш `src/lib`

.rs:

```
use std::mem::transmute;

pub struct Args {
        init: u32,
        by: u32,
        descr: String,
}

impl Args {
        pub fn new(init: u32, by:
            u32, descr: String) ->
            Args {
             Args{
                 init: init,
                 by: by,
                 descr: descr,
             }
        }
}

#[no_mangle]
```

```
20 pub extern fn get_struct() -> *
       mut Args {
21     let my_struct = unsafe {
           transmute(Box::new(Args::
           new(42, 1, "hi from Rust!"
           .to_string()))) };
22     my_struct
23 }
```

Здесь мы добавляем к нашей старой доброй `Args` еще одно поле с типом `String`, в которое мы будет записывать строку.

Затем мы создадим имплементацию для более каноничного создания нашей структуры (без этого можно обойтись, но неправильно это как-то). Создадим метод `new`, на вход которого будем передавать два числа и строку. Ну а дальше в функции просто создадим указатель на нашу структуру и вернем его перловому коду.

Обратите внимание, как создается объект `String`, в `Rust` есть два типа строк — `str` и `String`. В чем же отличия между `str` и `String`? `str` это так называемый строковый срез. Он имеет фиксированную длину и неизменяем, поэтому мы не можем использовать их в структуре (на самом деле можем, но с некоторыми затейливыми телодвижениями). Тип `String` представляет собой строку, размещаемую в куче и гарантированно являющуюся UTF-8 строкой. Обычно `String` получается из `str` путем преобразования через `.to_string`.

Обратное преобразование осуществляется через `&` — `str = &String`.

Более подробно об этом можно почитать в документации.

Теперь получим данные из `Rust`:

```
use v5.16;
package My::RustStr;

use FFI::Platypus::Record;

# Описываем структуру данных,
    которую мы ждем из библиотеки
record_layout(qw(
    uint    rs_counter
    uint    rs_by
    string  rs_descr
));

my $shared = 'target/release/
    libembed.dylib';

my $ffi = FFI::Platypus->new;
$ffi->lib($shared);
$ffi->lang('Rust');

# Связываем структуру с нашим
    пакетом MyRustStr и
    присваиваем ей алиас rstruct
```

```
20 $ffi->type("record(My::RustStr)"
       => 'rstruct');
21
22 # Аттачим библиотечную функцию
       get_struct в наш пакет
23 # и указываем, что ждем от нее
       объявленную выше структуру
24 $ffi->attach( get_struct => [] =>
       'rstruct', sub {
25   my($inner, $class) = @_;
26   $inner->();
27 });
28
29 package main;
30
31 # Используем наш пакет My::
       RustStr для работы с
       библиотекой
32 my $str = My::RustStr->get_struct
       ;
33
34 printf "Struct is [%d][%d][%s]\n"
       ,
```

```
35    $str->rs_counter,
36    $str->rs_by,
37    $str->rs_descr;
```

Сначала мы объявляем пакет `My::RustStr`, в котором описываем структуру данных, которую мы хотим от нашей библиотеки. Из-за особенностей модуля `FFI` запись `record_layout` может быть только одна на пакет (на самом деле нет, но лучше так не делать), поэтому мы выделяем нашу функцию в отдельный пакет.

Затем мы создаем новый тип данных в пакете `My::RustStr` `$ffi->type("record(My::RustStr)" => 'rstruct');` и называем его `rstruct`.

Далее мы, как обычно, создаем биндинг до библиотеки и эспортируем ее в пространство имен пакета `My::RustStr`.

Обратите внимание на `$ffi->attach( get_struct => [] => 'rstruct', sub {`, здесь в качестве возвращаемого значения функции мы указываем, что хотим получить нашу структуру `rstruct`. Также мы указываем кастомную функцию-обработчик, передавая coderef последним аргументом. В данном случае он не делает ничего кроме вызова библиотечной функции, но туда можно добавить какую-то дополнительную обработку до/после вызова внешней библитеки.

Затем мы просто распечатываем содержимое нашей структуры.

Теперь мы также можем передать структуру в нашу библиотеку без шаманства с pack и указателями. Изменим наш перловый код работы со счетчиком.

```
1 use v5.16;
```

```
2
3  package MyCounter;
4
5  use FFI::Platypus::Record;
6
7  record_layout(qw(
8      uint    rs_counter
9      uint    rs_by
10 ));
11
12 package main;
13
14 my $shared = 'target/release/
       libembed.dylib';
15
16 my $ffi = FFI::Platypus->new;
17 $ffi->lib($shared);
18 $ffi->lang('Rust');
19
20 # Создаем новую структуру данных
       с именем CounterStr
21 $ffi->type('record(MyCounter)' =>
       'CounterStr');
```

```
22
23 # Аттачим функции из библиотеки
24 $ffi->attach(createCounter => ['
       CounterStr'] => 'opaque');
25 $ffi->attach(getCounterValue => [
       'opaque'] => 'u32');
26 $ffi->attach(incrementCounterBy
       => ['opaque'] => 'u32');
27
28 # Создаем структуру, из которой
       мы создадим счетчик
29 my $counter = MyCounter->new(
30   rs_counter => 10,
31   rs_by => 2,
32 );
33
34 # Создаем объект счетчика и
       используем его для работы
35 my $ptr = createCounter($counter)
       ;
36 say getCounterValue($ptr);
37 say incrementCounterBy($ptr);
```

Здесь мы используем `opaque` в качестве возвращаемого значения функции `createCounter` так как нам нужен сырой указатель для работы с остальными подключенными функциями. Но при желании этот указатель можно превратить в структуру нужного формата с помощью функции cast.

В `Rust`-коде ничего менять не надо.

Запускаем:

```
alpha6$ perl ./plat.pl
10
12
```

Итого, мы создали новый объект счетчика и изменили его.

Теперь мы умеем работать с большей частью случаев, которые нам могут приго-

диться при использовании подключаемых библиотек (не только на Rust). Осталось научиться все это автоматически собирать при установке перлового модуля.

Для этого нам пригодятся Module::Build и Module::Build::FFI::Rust.

Приведем структуру проекта к виду:

```
Build.PL
ffi
lib
plat.pl
```

В lib у нас лежат Perl-библиотеки нашего проекта, в ffi — Rust-исходники. Build.PL выглядит так:

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

use strict;
use Module::Build;
```

```
use Module::Build::FFI::Rust;

my $build = Module::Build::FFI::Rust->new(
    module_name         => 'MyCounter::Record',
    dist_abstract       => 'FFI test',
    create_makefile_pl => 'traditional',
    dist_author         => 'Denis Fedoseev  <denis.fedoseev@gmail.com>',
    create_readme       => '0',
    license             => 'perl',
    configure_requires => {
    'Module::Build' => 0.38,
    'File::Which' => 0,
    'Module::Build::FFI' => '0.18'
  },
    build_requires      => {},
    requires            => {
```

```
        'perl'                  => '
           5.016000',
        'FFI::Platypus' => 0,
        'FFI::Platypus::Lang::
           Rust' => 0,

    },
    recommends              => {},
    recursive_test_files => 0,
    sign                    => 0,
    create_readme           => 1,
    create_license          => 1,
);

$build->create_build_script;
```

Обратите внимание на использование `Module::Build::FFI::Rust` вместо обычного `Module::Build`. Запускаем установку, подготавливаем `Build`-скрипт:

```
alpha6$ perl ./Build.PL
Can't find dist packages without
```

```
    a MANIFEST file
Run 'Build manifest' to generate
    one

WARNING: Possible missing or
    corrupt 'MANIFEST' file.
Nothing to enter for 'provides'
    field in metafile.
Created MYMETA.yml and MYMETA.
    json
Creating new 'Build' script for '
    MyCounter-Record' version 'v0
    .0.2'
```

Собираем наши модули:

```
alpha6$ ./Build
cargo build --release
    Updating registry `https://
        github.com/rust-lang/
        crates.io-index`
   Compiling winapi v0.2.5
   Compiling libc v0.2.4
```

```
    Compiling winapi-build v0.1.1
    Compiling advapi32-sys v0.1.2
    Compiling rand v0.3.12
    Compiling embed v0.1.0 (file
        :///Users/alpha6/projects/
        tests/build/ffi)
Building MyCounter-Record
```

Устанавливаем:

```
alpha6$ ./Build install
cargo build --release
Building MyCounter-Record
Files found in blib/arch:
    installing files in blib/lib
    into architecture dependent
    library tree
Installing /Users/alpha6/perl5/
    perlbrew/perls/perl-5.20.3/lib
    /site_perl/5.20.3/darwin-2
    level/auto/MyCounter/Record/
    Record.bundle
Installing /Users/alpha6/perl5/
```

```
perlbrew/perls/perl-5.20.3/lib
/site_perl/5.20.3/darwin-2
level/MyCounter/Record.pm
```

`MyCounter/Record/Record.bundle` это и есть наша `Rust`-библиотека, установленная в систему.

Теперь мы умеем работать с `Rust`-кодом из `Perl` и, самое главное, собирать из этого полноценные пакеты, которые и на CPAN выложить можно :)

В целом все, что описано в этой статье со стороны `Perl`, применимо к любому языку, который умеет изображать из себя C-библиотеку. Так что теперь вы всегда сможете взять нужную библиотеку, хоть на Java, и использовать в корыстных целях.

■ *Денис Федосеев*

7 Обзор CPAN за ноябрь 2015 г.

Рубрика с обзором интересных новинок CPAN за прошедший месяц.

Статистика

- Новых дистрибутивов — 154
- Новых выпусков — 778

Новые модули

HTTP::Cookies::PhantomJS

`HTTP::Cookies::PhantomJS` — расширение для `HTTP::Cookies`, которое позволя-

ет использовать файлы с куками формата, которые применяются в веб-скрапере PhantomJS. Это бывает полезно, когда куки устанавливаются в результате работы js-кода на стороне клиента при загрузке ресурса в PhantomJS, а последующие ресурсы можно загрузить, используя полученные куки, с помощью более легковесного веб-клиента, например, `LWP::UserAgent`.

GCCJIT

`GCCJIT` — это обвязка к библиотеке libgccjit, которая позволяет на лету формировать и компилировать код в машинные инструкции.

Locale::TextDomain::OO::Extract::Xslate

`Locale::TextDomain::OO::Extract::Xslate` позволяет извлекать строки для перевода из шаблонов `Text::Xslate` для последующего использования в `Locale::TextDomain::OO`.

DOM::Tiny

`DOM::Tiny` — это минималистичный HTML/XML DOM-парсер с поддержкой CSS-селекторов.

```
my $dom = DOM::Tiny->new(<<EOF);
<div>
  <p id="a">Test</p>
  <p id="b">123</p>
</div>
EOF
```

```
7
8  say $dom->at('#b')->text;   # 123
9
10 $dom->find('div p')->last->append
       ('<p id="c">456</p>');
11
12 say $dom->at('#c')->text;   # 456
```

App::remarkpl

`App::remarkpl` — это утилита, позволяющая запустить локальный веб-сервер на основе Mojolicious для отображения презентаций на основе remark. В качестве параметра указывается путь к файлу с презентации в формате Markdown, после запуска презентация может быть открыта в браузере по адресу localhost:3000

Devel::Trepan::Deparse

`Devel::Trepan::Deparse` — это плагин для отладчика `Devel::Trepan`, который добавляет две новые команды: *deparse* и *deval*. Команда deparse позволяет показать какой код будет выполняться в следующей инструкции, что бывает полезно, когда отладчик остановился на строке, в которой несколько операций и не ясно какая из них будет выполняться на следующем шаге:

```
1 if ( grep { $_ eq 'foo' } @bar )
      {...}
```

выражение внутри `if` или уже блок кода `grep` для какого-то текущего значения `@bar`? Команда deparse покажет какая именно операция будет выполняться следующей:

```
1 : deparse
```

```
# code to be run next...
$_
# contained in...
$_ eq 'foo'
```

Mail::DKIM::Iterator

Модуль `Mail::DKIM::Iterator` позволяет организовать проверку DKIM-записей в почтовых сообщениях и добавлять DKIM-подпись к ним. Основное отличие от `Mail::DKIM` — это возможность организовать неблокирующуюся потоковую обработку писем.

Unix::Pledge

Недавно появившийся в OpenBSD системный вызов pledge(2) теперь можно использовать в perl-программах, которые будут работать на этой платформе, с помощью модуля `Unix::Pledge`. Модуль позволяет наложить ограничения на процесс, например, запретить чтение или запись файлов, с указанием списка файлов-исключений.

Perl6::Export

`Perl6::Export` реализует Perl 6 роль `is export` для Perl 5 модулей, позволяя изящно экспортировать функции модуля. Например, сравнение с Exporter:

```
package My::Module;                    |
    package My::Module;
```

```
use Exporter;                    |
                                 |  use Perl6::Export;
                                 |
our @EXPORT = qw(foo bar);       |  sub foo is export(:MANDATORY);
                                 |
                                 |                                  sub
                                 |                                  bar
                                 |                                  is
                                 |                                  expo
                                 |                                  (:
                                 |                                  MAND
                                 |                                  );
                                 |
our @EXPORT_OK = qw(baz);        |  sub baz is export(:DEFAULT :
                                 |  qux);
our %EXPORT_TAGS =               |
    qw(qux => [qw(baz)]);        |
```

Pg::Reindex

Индексы Postgresql должны периодически пересоздаваться для достижения оптимальной производительности. Например, это можно сделать с помощью команды `REINDEX`, но в этом случае потребуется эксклюзивная блокировка всей таблицы.

Модуль `Pg::Reindex` выполняет создание индекса с помощью `CREATE INDEX CONCURRENTLY`, который может выполняться без блокировки. Затем он начинает транзакцию, в которой удаляет старый индекс и переименовывает новый. Таким образом достигается пересоздание индекса без длительной блокировки.

Обновлённые модули

Sereal 3.010

Вышло обновление (де)сериализатора `Sereal`. В новых выпусках исправлено множество багов, включая несколько случаев краха десериализатора, которые были обнаружены при тестирование с помощью фазера AFL.

JSON::Validator 0.63

Модуль `JSON::Validator` позволяет проводить проверку данных в формате JSON на соответствие JSON-схеме. В новой версии удалена поддержка использования модулей `YAML` и `YAML::Tiny` для загрузки

JSON-схемы в формате yaml (поддерживаются только `YAML::XS` и `YAML::Syck`). Связано это с тем, что `YAML` и `YAML::Tiny` в некоторых случаях некорректно выполняют разбор формата yaml.

PathTools 3.60

Вышло обновление дистрибутива PathTools, включающий модули для работы с путями на различных поддерживаемых в Perl платформах. В новой версии включён модуль `File::Spec::AmigaOS` для поддержки платформы AmigaOS.

MIME-Types 2.12

В новой версии дистрибутива `MIME-Types` обновлена база данных MIME-типов, в соответствии с текущим состоянием в IANA. Требования к версии Perl снижены до 5.6.

utf8::all 0.017

Прагма `utf8::all` тотально включает UTF-8: для исходного кода программы, перекодирует открываемые файлы и аргументы, переданные в командной строке, разрешает использование последовательностей `\N{...}` для задания юникод-символов по их имени и т.д. В новой версии появилась возможность отключать эффект прагмы, с помощью `no utf8::all`.

PDL 2.015

Вышел новый релиз PDL — модуля для работы научных вычислений и отображения данных. Как указано в журнале изменений — это отполированный релиз с исправлением всех замечаний обнаруженных в предыдущей версии с улучшенной поддержкой 64-битных целых.

perl 5.23.5

Вышла новая версия Perl 5.23.5 для разработчиков. В новом релизе проведена оптимизация, которая ускорила выполнение арифметических операций, таких как сложение, вычитание и умножение. Кроме того, теперь быстрее стали работать операции ++ и ——. Исправлено несколь-

ко ошибок с крахом интерпретатора, например, в функции pack: `perl -e 'pack "WH200000", \0'`

App::perlbrew 0.74

Обновился менеджер Perl-инсталляций perlbrew. В новой версии появилась поддержка переменной окружения `PERLBREW_LIB_PREFIX`, которая позволяет добавить каталог с библиотеками, имеющей приоритет над локальными каталогами библиотек perlbrew (т.е. `./lib:$PERL5LIB`). Теперь при загрузке файлов будет отдаваться предпочтение https-ресурсам и обязательно проверяться их сертификат.

podlators 4.00

Вышел новый мажорный релиз утилит для конвертирования POD-файлов. Теперь все входящие в дистрибутив модули используют единую версию. Релиз содержит большое количество исправлений, накопившихся с последнего релиза, вышедшего два года назад.

■ *Владимир Леттиев*

8 Интервью с Дмитрием Шаматриным

Дмитрий Шаматрин — программист, с недавнего времени организатор Perl-конференций

Как и когда научился программировать?

Во всей этой кухне я относительно недавно. Первые попытки были в школе, это был 2005-2006 год, и даже что-то получалось.

Хорошо помню, что Паскаль мне тогда очень не понравился. Но серьезно пригодилось программирование в процессе написания диплома в универе. Если кто не в курсе, по образованию я химик.

Так вот, есть такой большой и страшный аппарат — масс-спектрометр. Он определя-

ет состав вещества, разбивая молекулу на фрагментарные ионы, количество которых в единицу времени регистрируется масс-анализатором. По ним определяется состав. И вот если молекула большая, ионов будет много, и они будут разные. Их надо опознать и посчитать. Руками это делать долго и не интересно, особенно, если таких спектров несколько. Вот пришлось писать скрипт. Выбрал Perl, так как уже имел минимальный опыт с ним и примерно понимал, что да как.

Какой редактор используешь?

Emacs. Если/когда нет emacs, могу использовать vim. Тут уже дело привычки. Так сложилось, в основном, исторически. Долгое время под языки, с которыми я работаю, не было вменяемых IDE. А потом уже просто привык, и еще раз переучивать-

ся ради сиюминутного профита смысла для себя не вижу.

Как и когда познакомился с Perl?

Тут хочу сказать, что винда мне особо никогда не нравилась, да и сейчас не нравится. В 2006-2013 году моей основной системой был линукс, потом мак. Так вот.

С Perl я познакомился примерно в 2007 году. У нас в городе была локальная сеть, подключение ВПН в которой было не самой тривиальной задачей. Провайдеры тогда особо не заморачивались с поддержкой пользователей на линуксе. Пришлось всю автоматизацию подключения сделать на Perl.

С какими другими языками интересно работать?

Erlang, Haskell, Lua.

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

Не могу выделить одно преимущество, поэтому вот тот список, благодаря которому я еще пишу на Perl.

- Нечеловеческая гибкость.
- Возможность решать проблемы, как правило, малой кровью.
- Быстрая разработка.
- Вменяемое событийное программирование.
- Ссылочная механика.
- Здравая и продуманная система модулей/библиотек.
- Простота создания своих модулей (привет `ExtUtils::MakeMaker`).

Что, по-твоему, является самой важной особенностью языков будущего?

Перестать опускать планку порога вхождения в язык. Я прекрасно понимаю, что рынку не хватает программистов. Также я понимаю, что всем хочется программировать. Однако, это не повод упрощать инструменты до уровня, понятного и дебилу. Сложность перла практически оптимальная. По моему мнению, самое последнее скотство это жертвовать функциональностью во имя снижения порога вхождения. Целевая аудитория, как правило, не воспользуется, а вот урезанная функциональность печалит.

Что думаешь о Perl 6?

Отличный язык. Жаль, что невовремя.

Было бы приятно, если бы взлетело, но я не

понимаю двух вещей. Зачем это назвали Perl и сколько надо затратить сил и средств, чтобы вывести его сообщество хотя-бы на уровень Perl 5. Я думаю, что использовав другое имя для языка шансов взлететь было бы сильно больше.

Что лучше: работать на аутсорс или на продукт?

Тут нет однозначного ответа. Мне интереснее продукт, например, хотя, вроде как, по статистике, аутсорс стабильнее. Короче. Аутсорс — меньше рисков, низкая вероятность большого профита в будущем. Продукт — риски, которые, в основном, от тебя не зависят. Ну, например, бабло закончилось. В случае успеха можно получить некислую материальную выгоду.

Как жить программисту, если кажется,

что вокруг одни дураки?

Понять и простить. Правило 95% непобедимо.

Если же вопрос стоит о том, что делать с коллегами по цеху, которые по тем или иным причинам сейчас знают меньше тебя, то ответ прост и он единственно верный: обучать других и обучаться самому. Я, например, если ко мне обращаются с вопросами, стараюсь по мере сил помогать всем. Если люди учиться не хотят, то все плохо, и надо что-то менять. Вообще, чем выше суммарный уровень квалификации в хорошо слаженной команде, тем проще жить всем.

Лучше специализация или везде по чуть-чуть?

А это смотря что хочется делать и что делаешь. Если ты занимаешь позицию техлида и решаешь проблемы в определенном и ограниченном спектре задач (например, в телекоме), то тут сильно лучше иметь узкую специализацию. Если же твоя задача рулить разработкой большого проекта, то специализация ок, но куда важнее быстро разобраться в текущей задаче, а это требует везде по чуть-чуть. Особенно, это касается тех случаев, когда необходимо принимать серьезные технические решения, от которых много чего зависит или будет зависеть.

Так все-таки Erlang или Haskell?

Они решают примерно подобные задачи.

Эрланг позволяет довольно быстро построить приложение, которое будет устойчиво

к рантайм-ошибкам, которых, как все мы знаем, предостаточно всегда. Причем, 90% этих ошибок случается хрен пойми почему. К тому же, будучи функциональным языком, он позволяет делегировать некоторые сложные вещи виртуальной машине и сосредоточиться на реализации. Иммутабельные переменные, хорошая многопоточность. Ну и OTP. Все вместе это дает реально хороший, но при этом несложный язык, который очень хорошо работает под нагрузкой, редко падает и довольно неплохо масштабируется. Ну, если просто и на чистоту, то эрланг правильно использует раззвиздяйство программиста, культивируя три основные добродетели. Да и очень идеологически близок к перлу.

Хаскель позволяет делать практически то же самое, но при этом сильно сложнее и дороже, что требует больше ресурсов. Ну

и сложная система типов поначалу будет сильно взрывать мозг. Мой совет — учите оба :D Потому как понимание хаскеля, как чистого функционального языка, в конечном итоге, позволяет намного лучше писать на эрланге.

Сколько времени проводишь за написанием Perl-кода?

Когда как. Обычно несколько часов в день.

Стоит ли советовать молодым программистам учить сейчас Perl?

Стоит. То, что перл сейчас каждый год закапывают, а код на нем нечитаем и плох, есть следствие следующего забавного парадокса: в то время, когда появился перл, не было ничего, с чем бы он серьезно конкурировал в нише, например, сайтостроения.

К тому же, культуры программирования, как таковой, тоже, в широких массах, особо не было. Соответственно, весь тот код, за которые клеймят Perl, это не вина языка. Если бы в то же время появился, например, питон или руби, сейчас бы слава worn-языка была бы у одного из них. Просто не в то время. Почему стоит учить? Да потому, что он достаточно низкоуровневый, чтобы разобраться, как работают файловые дескрипторы, сокеты, счетчик ссылок. И он достаточно высокоуровневый, чтобы разобраться с концепциями в мире современного программирования. Ну и имеет очень большую базу библиотек, да.

Вопросы от читателей

Будут еще хакатоны и конференции?

Будут и много. Скорее всего что-то про-

ведем летом. Есть в планах одесская конференция по функциональному программированию.

Какой веб-фреймворк сейчас стоит использовать и почему?

Мне в последнее время очень нравится перловый Catalyst. Почему? Удобный, много умеет, под него написано очень много модулей. Ну и еще очень немаловажный момент — адекватный мейнтейнер. Минусы — громоздкий.

■ *Вячеслав Тихановский*