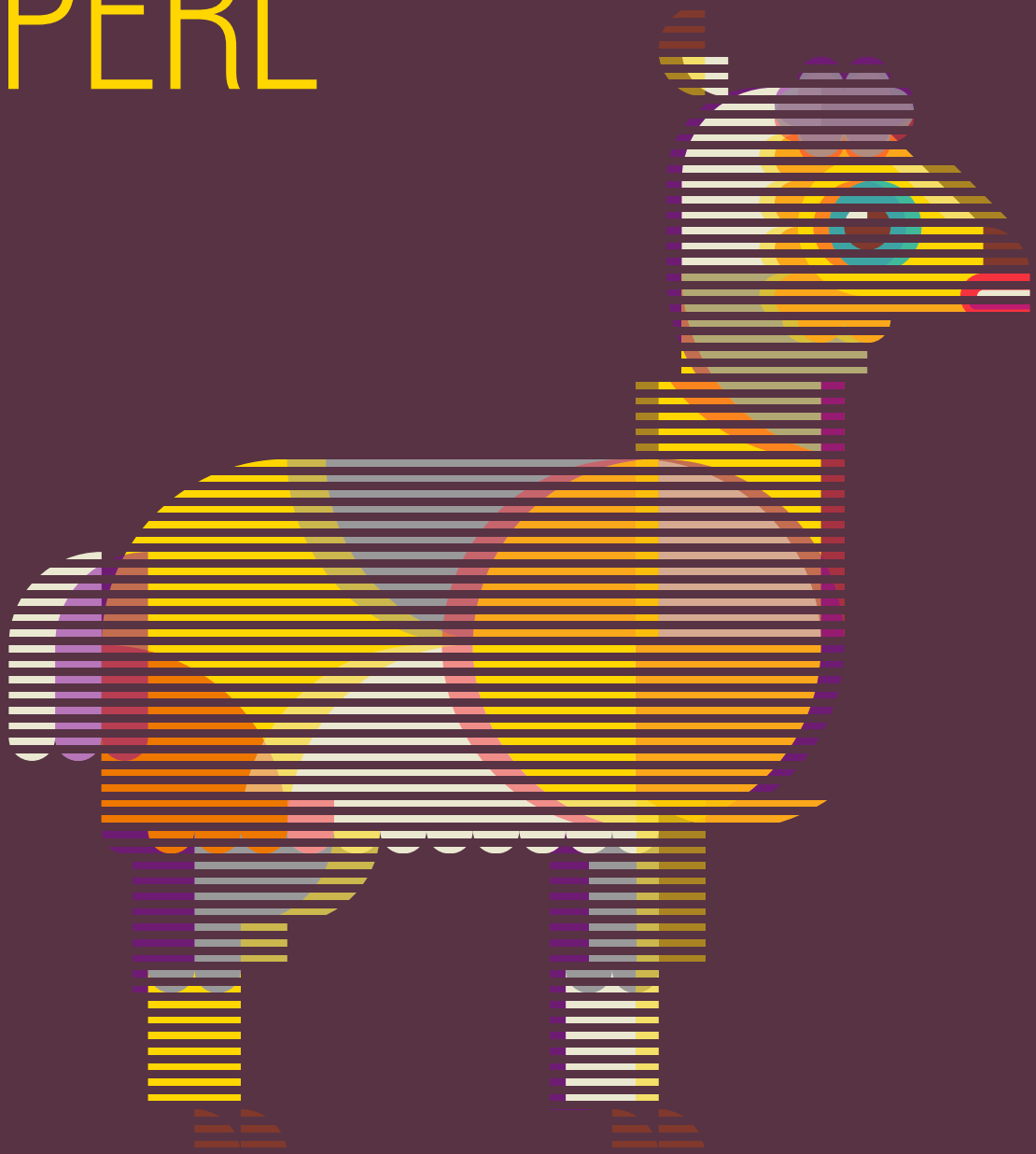
PRAGMATIC PERL

29



07/2015

pragmaticperl.com

Pragmatic Perl 29

pragmaticperl.com

Выпуск 29. Июль 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: Денис Федосеев, Наталья Савенкова, Владимир Леттиев

Обложка: Марко Иваныk

Корректор: Андрей Шитов

Выпускающий редактор: Вячеслав Тихановский

Ревизия: 2015-07-28 10:17

© «Pragmatic Perl»

Оглавление

1 От редактора

Напоминаем, что на днях (25-26 июля) состоится Perl-воркшоп в Одессе «Perl Mama» Не пропустите!

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 (R)?ex на практике

В жизни каждого человека настает момент, когда ему становится лениво делать одно и то же на удаленных серверах и хочется автоматизации

В решении этой тяжелой жизненной проблемы нам поможет (R)?ex. Rex представляет собой систему управления удаленными серверами и является аналогом Ansible, но, в отличии от него, написан на Perl и работает на всех платформах кроме Windows.

Первым делом установка. На момент написания статьи текущая версия на CPAN 1.2.1. Но устанавливать его лучше из репозитория пакетным менеджером вашего дистрибутива. Благо, сборки есть под все популярные дистрибутивы.

На данный момент основная проблема Rex — это слабая документация. Вроде все вещи описаны, но описаны где попало. Что-то есть на сайте, что-то на CPAN, что-то в списке рассылки — не очень удобно. Так что в этой статье не будет каких-то особо тайных знаний, в основном все просто будет сведено в кучу.

Первые шаги

Во первых, Rex можно использовать для простого запуска команд на удаленном сервере:

```
1 rex –H "example.com example.ru" –
      e "say run 'uptime'"
```

Здесь мы запускаем команду `uptime` на серверах example.com и example.ru и

получаем ее вывод в консоль.

Задачи в таком режиме выполняются последовательно, так что придется подождать, пока опросятся все сервера, особенно если их много.

Запуск из консоли это вариант если надо единоразово посмотреть: что там на серверах творится. Но постоянно набирать команды в консоли это решение для сильных духом и памятью. Мы же, как настоящие программисты, будем осваивать командные файлы.

По умолчанию Rex использует файл под названием Rexfile (удивительное совпадение).

Попробуем сделать что-то полезное.

Проверка синтаксиса

Первым делом укажу самое главное при написании командных файлов: проверка синтаксиса выполняется командой `rex –T`.

При ее выполнении будет выведено примерно следующее:

```
bash–3.2$ rex –T
Tasks
 checkout
 deploy     Deploy mysite
 sync
Server Groups
  myservers
      server1
```

В принципе, тут все очевидно. В случае ошибки синтаксиса появится обычное перловое сообщение об ошибке:

```
bash-3.2$ rex -T
syntax error at Rexfile line 22,
   near ""checkout" sub "
syntax error at Rexfile line 27,
   near "}"
Rexfile had compilation errors.
[2015-06-04 10:03:58] INFO -
   Exiting Rex...
[2015-06-04 10:03:58] INFO -
   Cleaning up...
```

Тоже все достаточно очевидно.

Режим отладки

В случае если с синтаксисом все правильно, но все равно ничего не работает — можно использовать режим отладки. Вызывается указанием `-d` в параметрах задачи.

```
rex -d stalled-task
```

Выведет гигантскую портянку со всеми действиями Rex, подставленными переменными и ответами удаленного сервера.

Теперь создадим Rexfile, который будет выполнять команду `deploy`. По этой команде Rex пойдет на удаленный сервер и сделает там `git pull` для сайта. Простейший вариант деплоя.

```
use Rex -feature => ['1.0'];

user "rex"; #Пользователь на удаленном сервере
private_key "~/.ssh/id_rsa";
public_key "~/.ssh/id_rsa.pub";
key_auth;

desc "Deploy the blog on example.ru";
task "deploy", "example.ru", sub {
  run "cd /srv/www/example.ru &&
```

```
        git pull origin gh-pages";
};
```

Теперь можно сказать: `rex deploy`.

Дальше Rex залогинится под пользователем `rex` с использованием указанных ключей и выполнит команду, указанную в опции `run`. Если пользователь на удаленном сервере совпадает с текущим, то первую секцию можно опустить. Будет использоваться текущий пользователь и его настройки ключей для ssh. Но если планируется использование команды через cron, то указывать надо обязательно, иначе ничего работать не будет. Так же при использовании через cron нужно указывать полные пути к ключам.

Это минимальный каркас командного файла для управления удаленным хо-

стом. Дальше можно добавлять модули, расширять и углублять задачи.

Во первых, Rex с одинаковым успехом может управлять группой хостов как одним. Для этого добавим описание группы:

```
group myservers => "server1", "
    server2", "serverN";
```

И приведем объявление задачи к виду:

```
task "deploy", group => "
    myservers", sub { ... };
```

Так как Rex использует DSL на основе Perl, то в объявлении серверов отлично работают перечисления, и наш список серверов можно было сделать в виде:

```
group myservers => "server[1..10]
    ";
```

В обычном режиме все операции с серверами будут производиться последовательно, что может занять достаточно приличное время. Для ускорения можно распараллеливать задачи, для этого служит опция `parallelism($count)` где `$count` это число тредов Rex. Каждый тред будет брать себе по серверу из списка и работать с ним. Правда, при этом смешивается вывод исполняемых команд (что логично), и определить что и где пошло не так, может быть сложно (если не настроить логгер и не использовать grep).

Чтобы выполнить команду локально, достаточно просто не указывать целевой сервер в атрибутах цели:

```
1 task "local_task", sub { ... };
```

Выполнит команду на хост машине.

Далее можно обновлять конфигурацию, копируя на хосты необходимые файлы (вы же не храните авторизационные данные в Git, правда?). Приведем содержимое раздела task к виду:

```
1  run "cd /srv/www/example.ru &&
       git pull origin gh-pages";
2  file "/srv/www/example.ru/test.
       conf",
3       source    => "files/test.
           conf";
```

Теперь при выполнении `rex deploy` будет выполнен `pull` из репозитория на удаленном сервере, а затем туда будет скопирован файл `files/test.conf`.

Синхронизация файлов

Таким же образом можно выкатывать все приложение, не используя `git pull` на удаленных хостах. Естественно, что делать это стоит не через описание всего приложения в секции file, а через старый добрый rsync.

Приведем наш Rexfile к виду:

```
use Rex -feature => ['1.0'];

# Подключаем модуль для работы с SCM, он поддерживает Git и SVN.
use Rex::Commands::SCM;
# Подключаем модуль для работы с rsync.
use Rex::Commands::Rsync;

user "rex"; #Пользователь на
```

```
    удаленном сервере
private_key "~/.ssh/id_rsa";
public_key "~/.ssh/id_rsa.pub";
key_auth;

# Объявляем репозиторий
set repository => "example",
        url => "git@github.com:spb
            -pm/spb.pm.git",
        type => "git",
        username => "username",
        password => "password";

desc "Deploy the site on example.
    ru";
task "deploy", "example.ru", sub
    {
     # Делаем checkout из
        указанного выше
        репозитория
     say checkout "example",
         branch => 'gh-pages',
         path => "site_repo";
```

```
26
27      # Заливаем файлы через rsync
           на удаленный хост
28      say sync "./site_repo/*", "/
           srv/www/example.ru", {
29              parameters => '--
                   backup --delete',
30      };
31  };
```

Теперь при выполнении `rex deploy` будет сделан git pull в директорию `site_repo`, в текущей директории и ее содержимое будет через rsync перенесено на удаленный сервер. В блоке `parameters => ...` задаются параметры rsync.

Через rsync можно получать и данные с удаленного сервера (может быть удобным для переодического получения логов и прочего в этом духе):

```
task "sync", "server01", sub {
   sync "/var/www/html/*", "html/
      ", {
    download => 1,
    parameters => '--backup',
   };
};
```

В принципе, указанного выше уже достаточно, чтобы написать свой logstash в 10 строчек (на самом деле нет).

Передача параметров в задачу

Еще полезным моментом может быть передача параметров в задачи. Например, можно указать ветку, если мы разворачиваем тестовое окружение.

```
rex deploy --branch=test-branch
   --parameter2=foo
```

Пример секции task:

```
task "deploy", "myserver", sub {
    my $params = shift;
    say $params->{'branch'};
    say run "uptime";
};
```

Собственно, стандартная перловая передача hashref с параметрами.

Обработка ошибок

Статус выхода команды `run` доступен в переменной `$?`. Так же для `run` можно включить режим `auto_die`. Что он делает, понятно из названия.

```
task "deploy", "myserver", sub {
    say run "cp /tmp/foo /tmp/bar
        ";
```

```
3       say run "...";
4 };
```

В случае отсутствия исходного файла для `cp` операция не выдаст ничего, и выполнится следующая команда. Исправим это поведение:

```
1 task "deploy", "myserver", sub {
2       say run "cp /tmp/foo /tmp/bar
           ", (auto_die => TRUE);
3       say run "...";
4 };
```

Теперь в случае ошибки мы получим красивый лог с описанием произошедшего.

sudo

Если нам нужно выполнить команду от root или пользователя, отличного от залогиненного, то надо использовать sudo:

```
sudo_password 'sudo password';

task "whoami", "example.ru", sub
    {
        say run "whoami";
        sudo {

                command => sub {
                        say run "
                           whoami
                           ";
                },
                user => 'myuser',
        };
};
```

Выведет 2 строчки:

```
1 rex
2 myuser
```

Ускоряем работу

Rex может кешировать инвентори для выполнения задач. Полезно, когда используется много вызовов `sudo` или других медленных операций. Для этого надо вызвать его с опцией `-c`:

```
1 rex -c deploy
```

После работы скрипта будет создан каталог `.cache`, в котором будет хранится инвентарь для последующих вызовов скрипта.

Управление конфигурациями

Теперь рассмотрим вещи, ради которых действительно стоит использовать системы управления конфигурациями. По большому счету то, что приведено выше, реализуется небольшим bash-скриптом, который все знают и умеют, и нет особого смысла осваивать Rex.

Модули

Для начала рассмотрим вопрос создания модулей. Нет смысла обсуждать, что это и зачем нужно, так что просто рассмотрим, как оно работает.

Модуль Rex фактически является стандартным модулем Perl. Соответственно,

большинство требований аналогично. Модели размещаются в каталоге `lib` директории, откуда выполняется `Rexfile`.

Для своих проектов я использую практически типовой набор ПО для создания окружения. Напишем модуль, который будет поднимать это окружение.

Создадим проект:

```
mkdir lib && cd lib && rexify
    MyRex::OS --create-module
```

В документации написано, что должна создаться директория `lib` с модулем и `Rexfile`, но на практике не создается.

```
lib
└── MyRex
        └── OS
            ├── __module__.pm
            └── meta.yml
```

Вот такое получится в результате работы этой команды. Файл `meta.yml` будет нужен только если вы собираетесь заливать свой модуль в репозиторий пакетов Rex. Файл `__template__.pm` — это файл нашего модуля.

Модуль будет устанавливать на сервер все необходимые пакеты и разворачивать окружение, приведем его к виду:

```
1  package MyRex::OS;
2
3  use Rex -feature => ['1.0'];
4
5  task "prepare" => sub {
6      sudo sub {
7          pkg [
8              "build-essential",
9              "supervisor",
10             "mariadb-common",
11                "git",
12             "vim",
```

```
13              ],
14          ensure => 'present';
15      };
16  };
17
18  1;
```

Команда `pkg` возьмет заданный список пакетов, проверит, что они установлены, и, в случае отсутствия какого-либо пакета — доустановит его.

Опция `ensure` говорит нам, что делать с пакетами:

- `present` — пакет должен быть установлен, какая версия, нам не важно;
- `latest` — должен быть установлен пакет последней версии, если это не так — пакет будет автоматически обновлен;

- `absent` — пакет *не* должен быть установлен, если пакет установлен — то он будет удален;
- `2.3.4` — должна быть установлена конкретная версия пакета.

Обратите внимание на использование `sudo` вокруг `pkg`, по умолчанию все операции выполняются от текущего пользователя, и самостоятельно `Rex` не будет переключаться в режим суперпользователя.

Создадим в корне проекта `Rexfile` вида:

```
use Rex -feature => ['1.0'];

user "rex";
password "password";
sudo_password "password";

group "srv" => "192.168.1.102";
```

```
8
9  require MyRex::OS;
10
11 task "prepare", group => "srv",
       sub {
12         MyRex::OS::prepare();
13 };
14
15 1;
```

В `Rexfile` мы объявляем группу для серверов и создаем команду `prepare`, которая будет выполнять команду `prepare` из модуля `MyRex::OS` на нужных нам серверах и в нужном окружении.

Теперь при запуске `rex prepare` на целевом сервере будут установлены все указанные выше пакеты.

Пришло время установить Perl.

Первым делом установим `perlbrew` и CPAN в системный Perl. Добавим в блок `sudo` в `prepare` нашего модуля:

```
append_if_no_such_line "/etc/environment",
    line  => "LC_ALL=en_US.UTF-8",
    regexp => [qr{^LC_ALL=}]; # this is an OR

run 'cpan -i App::perlbrew', (auto_die => TRUE);
```

Первая команда откроет `/etc/environment` и проверит, что там задан язык для `LC_ALL`. Если не задан, то выставит кодировку `en_US.UTF-8`.

Зачем это надо? По умолчанию эта переменная не проставлена, и многие ssh-клиенты ее не устанавливают. Соответ-

ственно, при установке `perl` мы получим кучу ругани и падение тестов, которые проверяют IO. На самом деле, для запуска задач через `Rex` эта проблема не актуальна, так как он выставляет `LC_ALL=C`, но при входе в консоль возможны варианты.

Вторая строка просто запускает `cpan -i`.

Установим Perl. Эту операцию я вынес в отдельный таск, т.к. там используется куча `run`, и простыня в `prepare` получается уж слишком большой.

```
1 task "install_perl" => sub {
2     say "Installing perl";
3     run "perlbrew init", (
          auto_die => TRUE);
4
5     append_if_no_such_line "~/.
          bashrc",
6             line  => "source ~/
                 perl5/perlbrew/etc
```

```
                /bashrc";
7
8           run "source ~/perl5/
                perlbrew/etc/bashrc";
9       run "perlbrew install perl
            -5.20.2", (auto_die =>
            TRUE);
10      say "Done";
11  };
```

Тоже ничего сложного, стандартный процесс установки из мануала `perlbrew`. В `append_if_no_such_line` мы просто дописываем строку, если ее не существует.

Вызов этого таска можно добавить *вне* блока `sudo` в `prepare` через `do_task "install_perl"`. Либо вызывать самостоятельно, по большому счету это выполняется один раз, и особой разницы нет.

Теперь мы можем подключать этот модуль в Rexfile любого нового проекта и получать единообразное окружение одной командой.

Ну и всякие мелочи напоследок.

say

Вывод команды say на больших логах выглядит чудовищно. Если сохранить вывод `run` в переменную и вывести его через `say`, то будет чуть менее ужасно. Но он настраивается:

```
sayformat('[%D] %h %s');


[2015-06-15 10:58:22]
    192.168.1.150 total 4
```

```
[2015-06-15 10:58:22]
    192.168.1.150 drwxr-xr-x 3
    root root 4096 Jun 15 09:22
    perl5
```

Локальные задачи

Если внутри таска на удаленных серверах надо выполнить задачу на хост-машине — используется функция LOCAL(&):

```
task "mytask", group => "srv" sub
    {
    # Это выполнится на удаленных
        серверах
    say run "uptime";

    # А это на локальном
    LOCAL {
      say run "uptime";
    };
```

```
9  };
```

profile

Важным моментом при использовании `Rex` являет тот факт, что он подключается к серверу в режиме nologin. В этом режиме не подгружаются файл `.bashrc` и иже с ним. В данной ситуации есть три варианта действий:

- делать `source ~/.bashrc` в команде;
- вручную задать все нужные переменные окружения в Rexfile;
- выставить настройку `source_global_p` `(0|1)`.

Настройка делает то же что и первый пункт списка, но автоматически.

Вывод команды при ошибке

По большей части нам все равно, что пишут команды в консоль при выполнении, не все равно становится, когда что-то ломается. Но каждый раз переключать с `run "uptime"` на `say run "uptime"` не интересно и лениво. Для решения этой проблемы есть специальная команда — `last_command_output`

```
task "mytask", group => "srv" sub {
    run "command-with-error";

    if ($?) {
        say last_command_output();
    }
```

```
6       }
7   };
```

Для версий старше 1.2 — надо будет изменить заголовок файла на:

```
1 use Rex –feature => ['1.3', 'tty'
      ];
```

поскольку в режиме notty не происходит перехват STDERR, а режим tty был отключен по умолчанию в каком-то из релизов 1.0+.

Заключение

В целом, `Rex` является очень мощным инструментом прямо из коробки. Он позволяет обойтись без строительства велосипедов, и при этом получить полный контроль над

управлением инфраструктурой самых разных масштабов.

Единственное что он пока не умеет — управление хостами на Windows, но если это не требуется — то дорога в мир Continuous Delivery для вас открыта :)

■ *Денис Федосеев*

3 Инструменты для поиска зависимостей в скрипте

Практическая статья про инструменты анализа зависимостей

В этот раз хочется немного рассказать про инструменты, которые я использую, когда нужно найти последовательность загрузки модулей или виновника, который загружает что-то лишнее. Не всегда бывает очевидно, не всегда grep способен решить проблемы.

Кроме того, такие инструменты дают общее представление, откуда и какие именно модули подгрузились для выполнения задачи. И, к удивлению, иногда их бывает значительно больше, чем ожидаешь, а порой там можно найти самые неожиданные

вещи.

Предыстория. Была у меня как-то очень простая задача — расширить функциональность программы, используя уже готовую библиотеку. Задача решилась, но скорость старта (не работы, а именно старта) упала до уровня, заметного человеку. Это очень подозрительно, если знаешь, что объективных причин на это нет. Не стоит игнорировать такие симптомы.

(Все представленные ниже примеры выдуманы из пальца и упрощены до предела для наглядности.)

Например, есть вот такая программа:

```
1 use strict; use warnings; use
    feature qw/say/;
2 use lib::abs qw/./;
3
```

```
# do something cool
say 'OK';

exit(0);
```

Она делает что-то полезное и важное, но мы хотим сделать ее еще полезнее, а нужное уже есть в другой библиотеке `Utils`. Подключи и используй. Пусть это будет такая библиотека Utils:

```
package Utils;
use strict; use warnings; use
    feature qw/say/;
use Common;

use Exporter;
our @ISA = qw(Exporter);
our @EXPORT_OK = qw(a);

sub a {
    say 'a';
}
```

```
12
13 1;
```

Любой нормальный человек подключает эту библиотеку и наслаждается жизнью с готовой функциональностью.

```
1 use Utils qw/a/;
2 a();
```

```
1 $ perl main.pl
2 a
3 OK
```

Для повышения запутанности связей модулей я сделала еще такой Common:

```
1 package Common;
2 use Abc;
3 use DateTime;
4 1;
```

и **Abc**:

```
package Abc;
use DateTime;
1;
```

Пример построен таким образом, что в результате формируются следующие зависимости:

```
main.pl
|-- Utils
    |-- Common
        |-- Datetime
        |-- Abc
            |-- Datetime
```

В реальности вам ничего о них неизвестно, поэтому представим, что мы о них не знаем.

perl -d

Если вы ничего не знаете о –d, тогда вам сюда: http://perldoc.perl.org/perldebug.html. Perl debugger отлично показывает все загруженные модули. Давайте начнем с общего списка:

```
$ perl –d main.pl
...
main::(main.pl:5):  a();

  DB<1> M

'Abc.pm' => '/home/wwax/Desktop/
   pragmatic/2/Abc.pm'
'Carp.pm' => '1.29 from /usr/
   share/perl/5.18/Carp.pm'
'Class/Singleton.pm' => '1.5 from
    /usr/local/share/perl/5.18.2/
   Class/Singleton.pm'
'Cwd.pm' => '3.40 from /usr/lib/
```

```
    perl/5.18/Cwd.pm'
'DateTime.pm' => '1.19 from /usr/
    local/lib/perl/5.18.2/DateTime
    .pm'
'DateTime/Duration.pm' => '1.19
    from /usr/local/lib/perl
    /5.18.2/DateTime/Duration.pm'
'DateTime/Helpers.pm' => '1.19
    from /usr/local/lib/perl
    /5.18.2/DateTime/Helpers.pm'
```

Команда M показывает список всех загруженных модулей. Но подождите! Я же только подключила Utils, а там нет никакого DateTime! Он мне вообще не нужен, поэтому надо бы избавиться, чтобы не притягивать лишнего.

Смотрим, где он у нас подключается.

```
$ perl -d main.pl
...
```

```
  DB<1> b load /usr/local/lib/
     perl/5.18.2/DateTime.pm
Will stop on load of '/usr/local/
   lib/perl/5.18.2/DateTime.pm'.

  DB<2> R
'/usr/local/lib/perl/5.18.2/
   DateTime.pm' loaded...

DateTime::CODE(0x1e58ce8)(/usr/
   local/lib/perl/5.18.2/DateTime
   .pm:9):
9:  our $VERSION = '1.19';

  DB<2> T
@ = DB::DB called from file '/usr
   /local/lib/perl/5.18.2/
   DateTime.pm' line 9
$ = require 'DateTime.pm' called
   from file '/home/wwax/Desktop/
   pragmatic/2/Abc.pm' line 2
....
```

`b load /usr/local/lib/perl/5.18.2/DateTime.pm` — установили точку останова на момент, когда будет выполняться первая инструкция этого модуля. Имя файла беру из списка загруженных модулей, полученных по команде `M`. `R` — перезапуск дебаггера, который затем остановился на строке 9 файла DateTime.pm. `T` — трассировка вызовов. Во второй строке видно, что:

```
require 'DateTime.pm' called from
    file '/home/wwax/Desktop/
  pragmatic/2/Abc.pm' line 2
```

наш модуль подключается в другом модуле `ABC`, в строке 2.

Это отличный инструмент!

Закомметируем здесь `#use DateTime;`. Ну просто посмотреть, что будет. Опять смот-

рим в `perl -d`, команда `M`. Но не может быть! Кто-то еще подключил DateTime.pm. Опять точка останова, опять трассировка. Только в этот раз DateTime подключил какой-то другой модуль с именем Common.pm... И так можно продолжать довольно долго, быстро надоедает.

Visually graphing

Есть вот такой невероятный инструмент: App::PrereqGrapher, о котором можно почитать тут.

Запустим:

```
$ prereq-grapher -nc main.pl
```

Эта команда сгенерировала файл `dependencie`
`.dot`, который можно превратить в `png`:

```
$ dot -Tpng dependencies.dot >
    output.png
```

Граф зависимостей в наших руках!

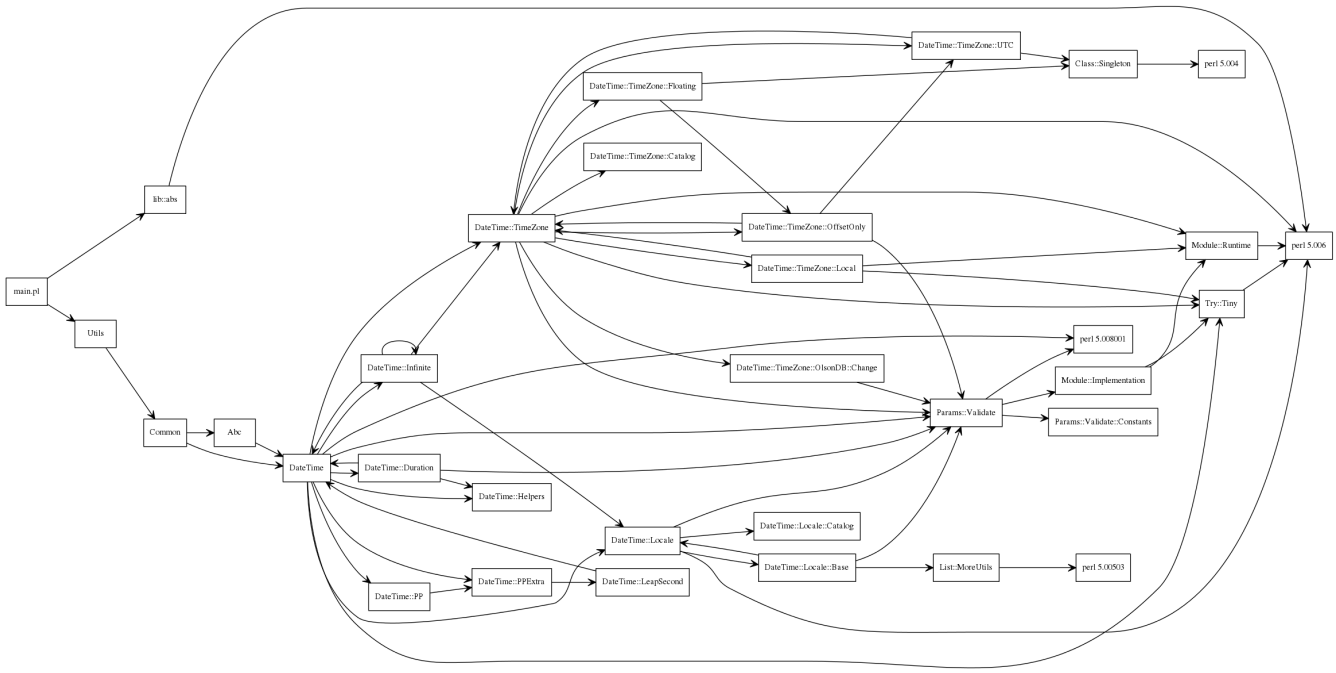


Рис. 0.1: Граф зависимостей

На графе очевидно, какой из модулей подключает DateTime, и сколько всего он еще тянет за собой. А все ради использования одной только функции `a()`. Ужас! Этот граф построен с использованием

флага `-nc`, что значит — не выводить *core modules*.

Если ничего не помогает

Не так давно мне открылась мощь утилиты `strace` (понятное дело, что `grep` иногда дает ответ быстрее, но случаи бывают разные, и еще не всегда очевидно, где именно искать). Для меня, как человека, пишущего на C только в академическо-институтских целях, открылся новый мир средств отладки. (Если у меня здесь есть примеры неправильного использования, очень прошу мне о них сообщить.)

Запускаем и отправляем вывод в лог.

```
$ strace perl main.pl &> strace.log
```

Смотрим, какие вызовы обращались к чему-нибудь с DateTime:

```
$ cat strace.log | grep DateTime.pm -C 5 | less
stat("/home/wwax/Desktop/pragmatic/2/Abc.pm", {st_mode=S_IFREG|0664, st_size=30, ...}) = 0
open("/home/wwax/Desktop/pragmatic/2/Abc.pm", O_RDONLY) = 6
ioctl(6, SNDCTL_TMR_TIMEBASE or SNDRV_TIMER_IOCTL_NEXT_DEVICE or TCGETS, 0x7ffdd665dd60) = -1 ENOTTY (Inappropriate ioctl for device)
lseek(6, 0, SEEK_CUR)                   = 0
read(6, "package Abc;\nuse DateTime;\n1;\n", 8192) = 30
...
stat("/usr/local/lib/perl/5.18.2/
```

```
    DateTime.pm", {st_mode=S_IFREG
    |0444, st_size=122107, ...}) =
     0
open("/usr/local/lib/perl/5.18.2/
    DateTime.pm", O_RDONLY) = 7
```

Это, конечно, уже совсем тяжелый случай, если мы говорим о *perl*. Но средства отладки, на мой взгляд, это первые необходимые инструменты для упрощения своей жизни.

А как разрешать найденную нами избыточную зависимость — это уже зависит от конкретного проекта и ваших архитектурных взглядов. Успехов!

■ *Наталья Савенкова*

4 Обзор CPAN за июнь 2015 г.

Рубрика с обзором интересных новинок CPAN за прошедший месяц

Статистика

- Новых дистрибутивов — 187
- Новых выпусков — 775

Новые модули

Canary::Stability

Необычный модуль, от которого теперь зависят практически все модули Марка Ле-

манна, `AnyEvent`, `Coro` и другие. Модуль используется на фазе конфигурации (при запуске `Makefile.PL`) и проверяет версию Perl, под которой происходит сборка. Если версия Perl 5.22.0 или старше, то модуль сообщает о том, что эта версия Perl не поддерживается.

Таким образом выражается протест против регулярного нарушения обратной совместимости текущей командой разработчиков Perl5 вопреки действующей политики perlpolicy.

Assert::Conditional

`Assert::Conditional` — это модуль, который позволяет организовать проверку выполнения определённых условий. Для программистов, знакомых с языком C, это

аналог assert. Работа модуля может управляться с помощью переменной окружения ASSERT_CONDITONAL, которая позволяет включать и отключать работу модуля, что полезно при работе в режиме отладки.

Список тестирующих функций просто огромен. Например:

```
# проверка на вызов в списочном контексте
assert_list_context()

# аргумент неопределён
assert_undefined(ARG)

# первый аргумент фукнции является blessed
assert_object_method()

# проверка, что передано три аргумента
assert_argc(3);
```

```
12
13 # строка является бинарной (
       кодовые точки от 0 до 255)
14 assert_bytes(ARG)
```

Algorithm::BloomFilter

`Algorithm::BloomFilter` — это XS-реализация вероятностной структуры данных Фильтра Блума. Подобные структуры данных используются в основном для систем хранения, позволяя быстро определить, существуют ли данные, избегая затратных операций обращения к диску.

DBD::SQLcipher

`DBD::SQLcipher` — это модуль, исходный код которого базируется на `DBD::SQLite` с тем лишь отличием, что вместо SQLite используется SQLCipher — расширение SQLite, которое обеспечивает прозрачное шифрование данных с помощью алгоритма AES-256. API модуля идентично `DBD::SQLite`, требуется лишь представить драйверу ключ (де)шифрования:

```
1 my $dbh = DBI->connect( "dbi:
    SQLcipher:dbname=/path/to/db.
    file", '', '' );
2 $dbh->do( qq{pragma key ="
    secret_db_password";} );
```

WWW::Telegram::BotAPI

`WWW::Telegram::BotAPI` — это модуль, который реализует доступ к API сервиса обмена сообщениями Telegram. Модуль использует либо `Mojo::UserAgent`, либо `LWP::UserAgent`, обеспечивая хорошую интеграцию с Mojolicious, но при этом не имея жёсткой зависимости от него.

Crypt::JWT

`Crypt::JWT` — это реализация стандарта JSON Web Token (RFC7519) для Perl. JWT в основном используется для реализации аутентификации пользователей в веб-окружении и технологии единого входа.

Обновлённые модули

Getopt::Long 2.47

За июнь вышло два обновления `Getopt::Long`, в которых исправлено две ошибки: формат числа с плавающей точкой теперь поддерживает запись без цифры перед точкой, например, `.2` (вместо `0.2`), использование ООП-интерфейса больше не нарушает последующую работу функционального.

Plack 1.0037

В новом релизе суперклея для веб-фреймворков и веб-серверов Plack модуль валидатора запросов и ответов `Plack::`

`Middleware::Lint` теперь поддерживает значение `HTTP/2` для переменной окружения `SERVER_PROTOCOL`.

Net::SSLeay 1.70

Вышел новый релиз модуля `Net::SSLeay`. Теперь модуль поддерживает сборку с OpenSSL 1.0.2 и 1.0.2a, а также теперь полностью совместим в LibreSSL.

Dancer2 0.160003

Вышел новый релиз веб-фреймворка Dancer2, в котором добавлено экранирование символов в идентификаторах для файловых сессий. Поскольку строка идентификатора использовалась как

часть пути к файлу сессии, использование последовательности `../` в идентификаторе, получаемого из cookie, позволяло получать доступ к произвольному файлу. Также произведён переход с использования модуля `HTTP::Headers` на более быстрый `HTTP::Headers::Fast`.

Dancer 1.3138

Также, как и в Dancer2, выполнено исправление в `Dancer::Session::YAML`, чтобы исключить возможность получения доступа к произвольным файлам через специально сформированный идентификатор файловой сессии в cookie.

Mango 1.18

Обновлён модуль `Mango` — реализация неблокирующегося драйвера для MongoDB. В новой версии появилась поддержка аутентификации `SCRAM-SHA-1`. Также исправлена ошибка в проверке BSON-идентификаторов. Дело в том, что для этого использовалось регулярное выражение:

```
$oid !~ /^[0-9a-fA-F]{24}$/
```

Проблема в том, что символ `$` приводит к совпадению не только в конце строки, но и перед символом переноса `\n` в конце строки. Таким образом, идентификатор с завершающим символом переноса строки проходил валидацию в драйвере, но приводил к ошибке в базе MongoDB. Вместо `$` теперь применяется `\z`.

Данная ошибка почему-то отмечена как проблема безопасности, которая позволяет организовать DoS-атаку на сервер MongoDB. Но Mango не производит переподключений или пинг сервера при ошибках, в отличие от ruby-реализации Moped, где подобная ошибка действительно позволяла организовать DoS-атаку.

HTTP::Headers::Fast 0.18

Новый релиз `HTTP::Headers::Fast` восстанавливает совместимость с `HTTP::Headers` (реализован отсутствовавший метод `content_type_charset`).

■ *Владимир Леттиев*

5 Интервью с Дэвидом Голденом

Дэвид Голден (David Golden) — автор многих модулей на CPAN, текущий разработчик официального драйвера MongoDB

Когда и как научился программировать?

Кажется, мне было девять лет, когда в нашем классе появился TRS-80, и я начал пробовать BASIC. Вскоре у меня дома появился собственный TI-99/4A, на котором я продолжил программировать на BASIC. Я уверен, что в том время мне это казалось чем-то серьезным, но я даже не представляю, чем я занимался.

В старших классах я программировал на Pascal и начал на C. В университете я много работал с C, а затем на последнем курсе

начал пробовать C++ и Java.

Какой редактор используешь?

В университете я познакомился с emacs, потому что все мое окружение им пользовалось, но он никогда не был чем-то большим; я ничего не кастомизировал и не программировал.

В конце девяностых я стал играться с Linux, для настройки мне нужен был vi. Я устал от того, что мне приходилось каждый раз вспоминать, как им пользоваться, и я решил использовать vim как основной редактор на Linux. В конце концов я пересилил себя и установил gvim на Windows на моем ноутбуке.

Vim мне особенно подошел, потому как я иногда испытываю боли в руках, а в

vim их можно почти не двигать; я полностью отключил стрелки на клавиатуре, и использую `CTRL-[` практически с той же частотой, что и `esc`.

Когда и как познакомился с Perl?

Я познакомился с Perl в конце девяностых, когда администрировал свой собственный микропровайдер для друзей и семьи, это был Linux, подключенный к одной из первых DSL-линий в США. Я пытался расшифровать программу “logwatch”, и с тех пор меня не оторвать.

В ранние годы я проводил много времени на Perlmonks. Читая, а затем и отвечая на вопросы, я преодолел фазу новичка.

Использование Perl на Windows вовлекло меня в CPAN Testers (хотелось упростить

отправку ответов о сломанных модулях на Windows). С тех пор я все больше погружался в Perl-разработку и затем в ядро.

С какими другими языками интересно работать?

Большинство моей работы я пишу на Perl, C и shell. Я испытываю любовь и ненависть к C. Часто программировать на нем невыносимо, но возможности, которые он дает по представлению данных и их обработке, вновь возвращают мою любовь.

Сейчас я увлечен Go, хотя в основном у меня была возможность только поиграться. Он подходят для моего мозга практически так же как и Perl, а скорость (по сравнению с Perl), модель распараллеливания и статическая компиляция решают некоторые

слабости Perl, которые меня раздражают.

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

Мне кажется, что доминантным преимуществом Perl является его возможность из «спичек и желудей» собрать программу. Он отлично масштабируется от однострочников, небольших скриптов до небольших систем, всегда можно решить с тем же опытом задачу большую и маленькую.

В некоторых случаях (но не во всем) я думаю, что философия TIMTOWDI (есть больше одного способа сделать это — *прим. перев.*) сильно недооценена и ведет к постоянным эволюциям идей.

Например, однажды я объяснял людям не из Perl-сообщества, что ООП в Perl

реализуется с помощью фреймворков, как в других языка занимаются веб-программированием. Нужен полноценный фреймворк? Берите Moose. Нужно что-нибудь полегче? Берите Moo. Есть какие-то специальные требования? Отлично, есть десятки ООП-фреймворков на выбор.

Десять лет назад никто не знал про типажи и роли. Теперь это доминантная особенность ООП в Perl. Этого нет в языках, где «все включено». Это появляется, когда люди пытаются изобретать велосипеды, когда язык их не предоставляет.

Также должен добавить, что больше не считаю CPAN сильным преимуществом Perl, во всяком случае в относительном смысле. Он до сих пор очень полезен, но у многих языков такое же количество

библиотек. Стартапы предоставляют API вначале для «популярных» языков, а затем уже для Perl (если вообще такое случается), поэтому Perl обычно отстает в библиотеках для неожиданно ставших популярными сервисов.

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

Это может звучать заезжено, но, мне кажется, что «мастшабируемость» — самая важная вещь, но в разных направлениях.

Мне думается, что люди часто слишком фокусируются на масштабируемости в пределах одной системы, в частности на распараллеливании на многоядерных и многопроцессорных системах.

Но думаю, что масштабируемость в преде-

лах нескольких нод будет также важной. Под «облаком» уже многие понимают фермы, и популярность контейнеров будет только продвигать этот тренд быстрее.

Поэтому мне кажется, что у языков с разделяемой памятью выигрывают языки с хорошо спроектированной моделью передачи сообщений. Хотя они и теоретически одинаковы, на практике, как мне кажется, разработчикам гораздо легче работать с сообщениями между несколькими нодами.

Также выигрывают языки с фермо-ориентированной разработкой. Статическая сборка или простота развертывания будет ключевым преимуществом.

Другой интересной стороной мастшабирования являются люди. У некоторых языков есть репутация (оправданная или нет)

удачного (или неудачного) применения для больших команд. Лучшие языки должны быть одинаково хороши как для одного разработчика, так и для большой команды.

Что думаешь о Perl 6?

Мое мнение не сильно изменилось с моего февральского поста:

> Мне до сих пор не особо понятно, что является «киллер фичей». Если Perl 6 будет намного быстрее Perl 5 (в абсолютной производительности или в распараллеливании), то он будет достаточно знаком разработчикам Perl 5, которые смирятся со сложностью изучения языка в обмен на производительность.

Но я не вижу ничего такого, что было бы интересно другим сообществам или новичкам. Объем и сложность языка слишком удручают. Для контраста, Go — небольшой, крайне своевольный язык — стал широко популярен в определенных нишах. Он решает некоторые серьезные проблемы и привлекает как энтузиастов статических, так и динамических языков.

Мне кажется, что разработчики Perl 6 должны найти несколько ключевых особенностей языка, которые действительно имеют преимущество над другими. Не «хитрые фишки» — в продолжение возможности делать сложные вещи просто в

обычной жизни.

Ты поддерживаешь так много модулей. Как это возможно?

Во-первых, я пишу много модулей! Я развил в себе привычку писать все как многоразовые, независимые библиотеки. Это заставило меня автоматизировать весь процесс разработки модуля.

Например, у меня есть одна шелл-функция, которая строит скелет Dist::Zilla и создает публичный и приватный репозитории. Это позволяет мне перейти от идеи к реализации за несколько секунд.

Во-вторых, я стараюсь автоматизировать все, что могу и для минимазации времени, которое я трачу на рутинную работу, и для предотвращения повторяемых оши-

бок. Использование Dist::Zilla для этого радикально изменило то, как я выкладываю код. Все, что мне требуется для выкатки модули, это убедиться, что файл Changes имеет актуальную версию. Все на автопилоте, включая версии модулей, тестирование перед релизом, генерацию Pod-документации, git-теги и прочее.

Также я использую утилиту git-hub от Инги (Ingy döt Net), которая помогает управлять комментариями и pull-запросами целиком из командной строки. Я могу выкачать pull-запрос, выполнить rebase, протестировать его, закрыть pull-запрос с комментарием и выкатить новую версию с помощью Dist::Zilla за несколько минут не выходя из терминала.

Кто написал первую версию HTTP::Tiny, почему был нужен этот модуль?

Еще в 2010 г. несколько людей на p5p обсуждали проблему, что Perl 5 не может запустить CPAN по HTTP без зависимости от таких командных утилит как curl или wget. Мы подумывали о добавлении HTTP::Lite в ядро, но у него было несколько зависимостей, и он выглядел не очень поддерживаемым.

В ответ на это Кристиан Хансен (Christian Hansen) написал первый черновой вариант HTTP::Tiny. Задачей было получить HTTP-клиент, который был бы достаточным для работы CPAN.pm по HTTP и способным скачать LWP.

После первого чернового варианта я начал помогать работой над API и рефакторингом внутренностей. Большинство кода все еще от Кристиана, даже если git blame выдает меня, но это из-за рефакторинга.

У нас было отличное сотрудничество. Мы стараемся обсуждать все важные изменения перед их реализацией, и мне кажется, что это сохранило хороший баланс между возможностями и минимализмом. Я был очень удивлен (приятно), что этот модуль для некоторых стал альтернативой LWP.

Как думаешь, стоит ли в ядро включить больше ::Tiny-модулей?

Не только потому что они ::Tiny. Оригинальной идеей ::Tiny-модулей было то, что они выполняют 90% работы, используя 10% размера/памяти некоторых популярных модулей и зависят только от ядра Perl. Они задумывались для простой установки, поэтому они не обязательно должны быть в ядре.

Почему важны QA-хакатоны?

Большинство проблем в мире сборки Perl и тестирования тяжело решаемы в небольшие свободные промежутки времени, которые выпадают разработчикам в их ежедневной жизни. Или им нужно время для сосредотачивания, или нужно собрать нескольких людей для обсуждения как что-либо сделать. QA-хакатоны дают возможность и время разработчикам встретиться и двигаться дальше.

Я много пишу о том, как и почему работают QA-хакатоны в своих ежегодных опусах.

(Если вы хотите поддержать QA-хакатон в следующем году, вы можете пожертвовать на хакатон 2015 г., и средства перейдут на следующий год.)

Как участник p5p, что ты думаешь о stableperl?

Скажу от своего имени, не от имени p5p, я несколько разочарован, что опасения, которые привели к его созданию, не могли быть решены стандартным способом.

Но несмотря на это, я считаю это проект отличным экспериментом. Как, например, и rperl, любой эксперимент вокруг Perl 5 говорит о том, что есть спрос в сообществе на что-то, чего не дает Perl 5. Если у форка будет продолжение, это серьезный сигнал о том, что нужно пользователям.

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

Я работаю в MongoDB, где занимаюсь разработкой MongoDB Perl-драйвера. Это значит, что практически 100% моего времени я пишу на Perl (или XS).

В течение последнего года я работаю над Perl-драйвером следующего поколения, который является практически полностью переписанным оригинальным драйвером с улучшенной согласованностью, совместимостью, портируемостью и надежностью. Если верить git blame, то я ответственнен за 90% всего кода!

Я планирую выпустить релиз-кандидата в течение нескольких недель, поэтому если вы используете MongoDB, обратите внимание.

Стоит ли молодым программистам сейчас советовать учить Perl?

Мне кажется, что мы должны советовать учиться программировать, а язык не так важен, как важен получаемый опыт.

Когда я учился программировать, зеленый текст на черном фоне был крутой штукой, поэтому терминал-ориентированное программирование было довольно привлекательным. Сегодня, однако, ожидания современных детей гораздо выше, и они будут хотеть опыта в графической среде в качестве первого языка.

Для более взрослых, у меня смешанные чувства насчет динамического языка в качесте обучения, так как он может скрывать некоторые важные фундаментальные вещи, которые нужно знать. Но, возможно, это мое предвзятое мнение, так как я начинал с Pascal и C.

До сих пор работаешь с Perl на Windows?

Когда я работал консультантом, у меня был рабочий Windows-ноутбук, и я его исполь-

зовал как ежедневную Perl-платформу. Те мучения в конечном счете привели меня к CPAN Testers и помощи Адаму Кеннеди в создании Strawberry Perl.

Но как только у меня появилась возможность получить бесплатную виртуальную машину, я переключился на Linux в моей ежедневной работе с Perl. Большинство моих программ все еще учитывали существование Windows, но я перестал толкать камень на Windows-гору.

Теперь в работе с MongoDB я вынужден сохранять Windows-совместимость, поэтому мне приходится снова работать с Windows. Мне очень понравилась утилита berrybrew от Дэвида Фаррела, так как она облегчила тестирование на разных версиях Perl.

В какие игры играешь?

В основном сейчас я играю в настольные и карточные игры (оффлайн-игры). Недавно довелось попробовать Seven Wonders, и мне очень понравилось. Дома я в основном склоняюсь к играм, в которые играют мои дети. Часто это Ticket to Ride, также моим детям нравится Uno.

■ *Вячеслав Тихановский*