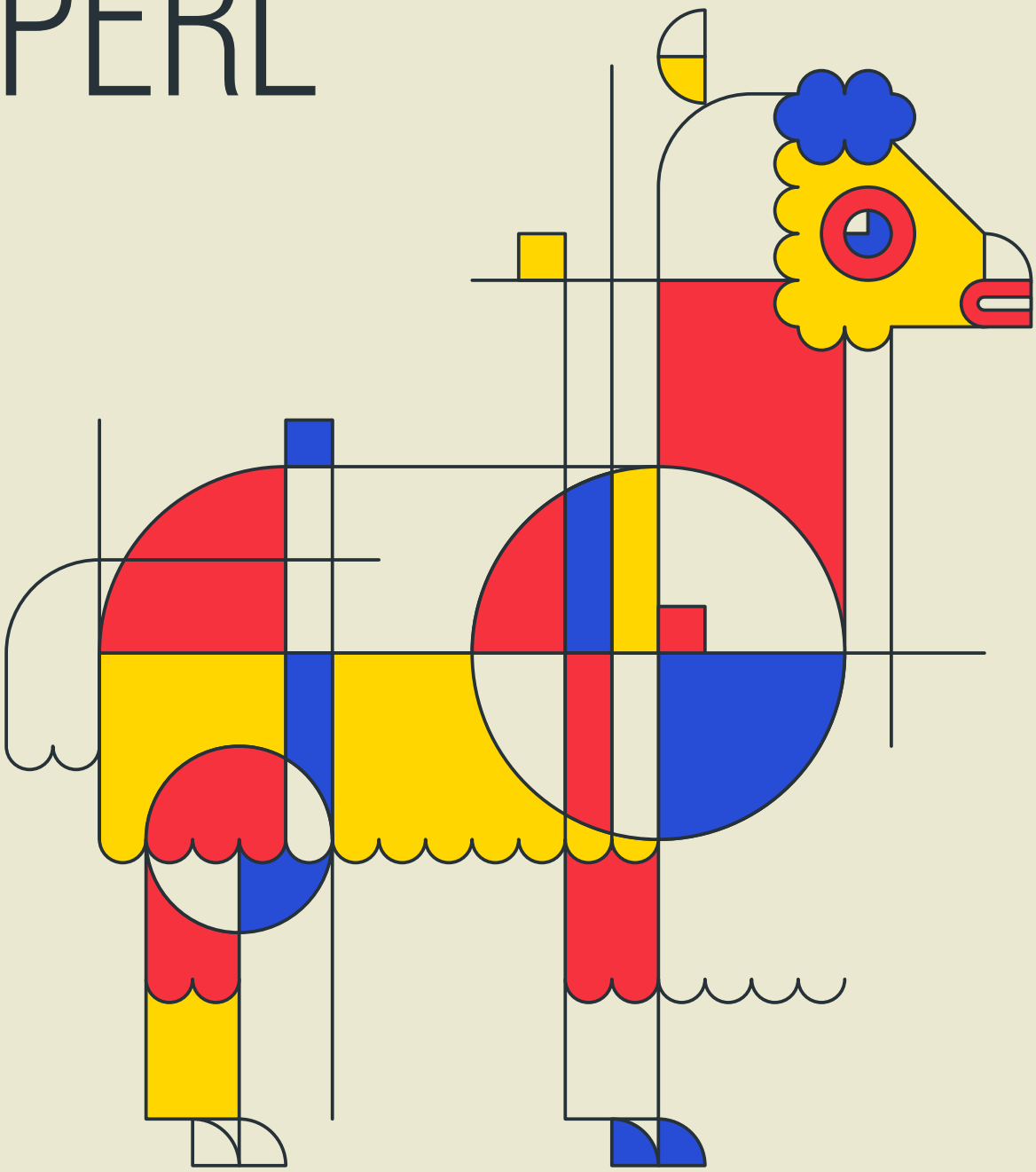
PRAGMATIC PERL 20



10/2014

pragmaticperl.com

Pragmatic Perl 20

pragmaticperl.com

Выпуск 20. Октябрь 2014

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: Вячеслав Коваль, Владимир Леттиев, Константин Токар

Обложка: Марко Иваныk

Корректор: Андрей Шитов

Выпускающий редактор: Вячеслав Тихановский

Ревизия: 2014-11-29 16:24

Оглавление

1. От редактора

В этом месяце выпуск несколько задержался. Но это зависело не совсем от нас и будет обязательно исправлено.

На прошлой неделе прошел австрийский Perl-воркшоп в Зальцбурге, на котором побывал Ларри Уолл, с ним можно почитать небольшое интервью.

В наших интервью мы обычно задаем вопросы от читателей. Через наш twitter сообщаем с кем будет интервью в следующем номере. Так можно задавать свои вопросы.

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

Приятного чтения.

■ *Вячеслав Тихановский*

2. Событийно-ориентированное программирование. Введение

Первая статья из цикла по событийно-ориентированному программированию

Данная статья будет первой в цикле статей по СОП (событийно-ориентированному программированию). Конечной целью данного цикла является понимание того, как работают EV и AnyEvent изнутри и как их применять более эффективно в реальных приложениях.

Первая часть цикла будет использовать C, чтобы показать, как работают:

- неблокирующий ввод-вывод (`open`/`fcntl + O_NONBLOCK` и ошибка `EWOULDBLOCK` (для сокетов));
- мультиплексирование ввода-вывода (`select`, `poll`, `epoll` (Linux 2.6+), `kqueue` (BSD), `eventfd` (Linux 2.6.27+), Input/output completion port (IOCP) (Windows NT 3.5+));
- как устроены библиотеки `libev`, `libuv` и `libevent` и как с ними работать;
- как устроен биндинг EV к libev;

Вторая часть цикла будет использовать Perl для того, чтобы описать работу модулей EV и AnyEvent, и, возможно, новый модуль UV.

Согласно Википедии, серверное приложение при использовании СОП (в нашем случае это AnyEvent+EV) реализуется на системном вызове, получающем события одновременно от многих дескрипторов (мультиплексирование). При обработке событий используются исключительно неблокирующие операции ввода-вывода, чтобы ни один дескриптор не препятствовал обработке событий от других дескрипторов. Далее рассмотрим мультиплексирование ввода-вывода.

Мультиплексирование ввода-вывода

Как сказано в книге Роберта Лава «Linux: системное программирование» (2-е издание, стр. 82), мультиплексированный ввод-вывод позволяет приложениям параллельно блокировать несколько файловых дескрипторов и получать уведомления, как только любой из них будет готов к чтению или записи без блокировки. По умолчанию файловые дескприторы (например, сокеты, каналы) блокируют выполнение процесса. Это означает, что когда мы вызываем на файловом дескрипторе функцию, которая не может выполниться немедленно, наш процесс переходит в “спящее” состояние и ждет, когда будет выполнено определенное условие. Для того, чтобы использовать неблокируемый ввод-вывод, необходимо использовать флаг `O_NONBLOCK` для функций open или fcntl. Пример неблокирующего ввода-вывода мы рассмотрим в следующей статье, когда будем рассматривать сокеты и клиент-серверные приложения.

Для того, чтобы понять, как работают функции для мультиплексирования ввода-вывода, рассмотрим простой пример — программа блокируется, дожидаясь поступления ввода на стандартный ввод (stdin), блокировка может продолжаться вплоть до 5 секунд. Эта программа отслеживает всего один файловый дескриптор, поэтому здесь отсутствует мультиплексный ввод-вывод как таковой. Однако данный пример должен прояснить использование этих системных вызовов:

Функция select()

```
#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

#define TIMEOUT 5     // Установка таймаута в секундах
#define BUF_LEN 1024  // Длина буфера считывания в байтах

int main() {
  // Объявляем структуру для определения
     продолжительности времени ожидания
  struct timeval tv;
  // Ожидаем не дольше 5 секунд
```

```
13  tv.tv_sec   = TIMEOUT;
14  tv.tv_usec  = 0;
15
16  // Объявляем набор дескрипторов для чтения
17  fd_set readfds;
18  // Дожидаемся ввода на stdin.
19  FD_ZERO(&readfds); // Сбрасываем все биты в наборе
       readfds
20  FD_SET(STDIN_FILENO, &readfds); // Устанавливаем бит
       для стандартного ввода (STDIN)
21
22  /**
23   * Блокируем процесс, пока не поступят данные на STDIN,
         либо пока не
24   * истечет время TIMEOUT
25   * В качестве первого параметра передаем максимальный
        номер дескриптора
26   * + 1 */
27  int ret = select(STDIN_FILENO + 1, &readfds, NULL, NULL
       , &tv);
28  if (ret == -1) {
29    perror("select");
30    return 1;
31  } else if (!ret) {
32    printf("%d seconds elapsed.\n", TIMEOUT);
33    return 0;
34  }
35
36  /**
37   * После возврата из функции select с помощью функции
        FD_ISSET проверяем,
38   * какие биты в наборе остались установленными
39   */
40  if (FD_ISSET(STDIN_FILENO, &readfds)) {
41    char buf[BUF_LEN+1];
42
43    // Блокировка гарантированно отсутствует
44    int len = read(STDIN_FILENO, buf, BUF_LEN);
45    if (len == -1) {
46      perror("read");
47      return 1;
48    } else if (len) {
49      buf[len] = '\0';
50      printf("read: %s\n", buf);
51    }
52
53    return 0;
54  }
```

```
55
56   fprintf(stderr, "Этого быть не должно!\n");
57   return 1;
58 }
```

Функция poll()

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <poll.h>
4
5  #define TIMEOUT 5     // Установка таймаута в секундах
6  #define BUF_LEN 1024  // Длина буфера считывания в байтах
7
8  int main() {
9    struct pollfd fds;
10
11   // Отслеживаем ввод на stdin
12   fds.fd = STDIN_FILENO;
13   fds.events = POLLIN;
14
15   // Выполняем блокирование
16   int ret = poll(&fds, 1, TIMEOUT * 1000);
17   if (ret == -1) {
18     perror("poll");
19     return 1;
20   }
21
22   if (!ret) {
23     printf("%d seconds elapsed.\n", TIMEOUT);
24     return 0;
25   }
26
27   if (fds.revents & POLLIN) {
28     char buf[BUF_LEN+1];
29
30     // Блокировка гарантированно отсутствует
31     int len = read(STDIN_FILENO, buf, BUF_LEN);
32     if (len == -1) {
33       perror("read");
34       return 1;
35     } else if (len) {
36       buf[len] = '\0';
37       printf("read: %s\n", buf);
38     }
39   }
40
```

```
41   return 0;
42 }
```

Функция epoll()

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/epoll.h>
5
6  #define TIMEOUT 5 // Установка таймаута в секундах
7  #define BUF_LEN 1024  // Длина буфера считывания в байтах
8  #define MAX_EVENTS 64
9
10 int main() {
11   /**
12    * Задаем контекст опроса событий
13    * На выходе получаем файловый дескриптор,
         ассоциированный с новым
14    * экземпляром epoll, который создается в функции
         epoll_create1().
15    * Данный файловый дескриптор не имеет отношения к
         реальному файлу; это
16    * просто указатель, который должен применяться с
         последующими вызовами,
17    * задействующими возможность опроса событий.
18    */
19   int epfd = epoll_create1(0);
20   if (epfd < 0) {
21     perror("epoll_create1");
22     return 1;
23   }
24
25   /**
26    * Добавляем файловый дескриптор STDIN_FILENO в
         контекст опроса epfd
27    */
28   struct epoll_event event;
29   event.data.fd = STDIN_FILENO;
30   event.events  = EPOLLIN;
31
32   int epctl = epoll_ctl(epfd, EPOLL_CTL_ADD, STDIN_FILENO
       , &event);
33   if (epctl) {
34     perror("epoll_ctl");
35     return 1;
36   }
```

```
37
38    /**
39     * Ожидаем событие чтения из STDIN_FILENO,
          ассоциированному с экземпляром
40     * опроса событий epfd
41     */
42    struct epoll_event *events = malloc(sizeof(struct
         epoll_event) * MAX_EVENTS);
43    if (!events) {
44      perror("malloc");
45      return 1;
46    }
47
48    int nr_events = epoll_wait(epfd, events, MAX_EVENTS,
         TIMEOUT * 1000);
49    if (nr_events < 0) {
50      perror("epoll_wait");
51      free(events);
52      return 1;
53    }
54    if (!nr_events) {
55      printf("%d seconds elapsed.\n", TIMEOUT);
56      return 0;
57    }
58
59    for (int i = 0; i < nr_events; ++i) {
60      if (events[i].events & EPOLLIN) {
61        char buf[BUF_LEN+1];
62
63        // Блокировка гарантированно отсутствует
64        int len = read(STDIN_FILENO, buf, BUF_LEN);
65        if (len == -1) {
66          perror("read");
67          return 1;
68        } else if (len) {
69          buf[len] = '\0';
70          printf("read: %s\n", buf);
71        }
72      }
73    }
74
75    free(events);
76
77    return 0;
78  }
```

Функция kqueue()

```c
#include <sys/types.h>
#include <sys/event.h>
#include <sys/time.h>
#include <unistd.h>
#include <stdio.h>

#define TIMEOUT 5
#define BUF_LEN 1024

int main() {
  // registration event
  struct kevent change; // events we want to monitor
  struct kevent event;  // event that were triggered

  // create event queue
  int kq = kqueue();
  if (kq == -1) {
    perror("kqueue");
    return 1;
  }

  EV_SET(&change, STDIN_FILENO, EVFILT_READ, EV_ADD, 0,
     0, NULL);

  struct timespec ts;
  ts.tv_sec = TIMEOUT;
  ts.tv_nsec = 0;

  while(1) {
    int nev = kevent(kq, &change, 1, &event, 1, &ts);
    if (new < 0) {
      perror("kevent");
      return 1;
    }
    if (nev == 0) {
      printf("%d seconds elapsed.\n", TIMEOUT);
      close(kq);
      return 0;
    }
    if (event.ident == STDIN_FILENO) {
      char buf[BUF_LEN + 1];

      int len = read(STDIN_FILENO, buf, BUF_LEN);
      if (len == -1) {
        perror("read");
        close(kq);
```

```
46          return 1;
47        } else if (len) {
48          buf[len] = '\0';
49          printf("read: %s\n", buf);
50          close(kq);
51          return 0;
52        }
53      }
54    }
55    close(kq);
56    return 0;
57  }
```

Я не буду расписывать, как работают функции для каждого примера, думаю по комментариям в коде должно быть все ясно. Лишь вкратце опишу, как работает epoll и kqueue, т.к. они описаны в книгах и статьях меньше всего.

Итак, в примере с epoll мы видим аналогию с AnyEvent — регистрация наблюдателя откреплятеся от самого акта наблюдения. Один системный вызов инициализирует контекст опроса событий (для epoll — вызов функции epoll_create1(), для AnyEvent - создание переменной состояния (condvar)), другой добавляет в контекст наблюдаемые файловые дескрипторы или удаляет их оттуда (для epoll — системный вызов epoll_ctl, для AnyEvent — создание наблюдателей для определенных событий (io, timer и т.д.)). Вызов метода send() в нашем случае означает выполнение события, что влечет за собой выход из колбэка и вызов метода recv()), третий выполняет само ожидание события (для epoll — системный вызов epoll_wait(), для AnyEvent — вызов метода recv() для переменной состояния).

Аналогичный пример на AnyEvent можете посмотреть в статье «Всё, что вы хотели знать об AnyEvent, но боялись спросить» из выпуска 1 данного журнала.

Системный вызов kqueue() аналогичен вызову epoll — сначала регистрируем фильтр событий с помощью функции kqueue(), затем с помощью макроса `EV_SET` настраиваем структуру change для отслеживания ввода на stdin, затем с помощью kevent отслеживаем события.

Помимо приведенных примеров создания таймера для блокирующего ожидания с помощью мультиплексирования ввода-вывода имеется еще два способа, как это сделать:

- использовать вызов функции alarm, которая генерирует сигнал `SIGALRM`, когда истекает заданное время;
- использование новых параметров сокета — `SO_RCVTIMEO` и `SO_SNDTIMEO` (специфична для сокетов).

Функции `eventfd` и `IOCP`, а также приведенные выше способы создания таймера будут рассмотрены в следующих статьях. В следующей статье будет создано приложение с использованием мультиплексирования ввода-вывода и неблокирующего ввода-вывода на нескольких дескрипторах с использованием сокетов.

Ссылки, где можно почитать про мультиплексирование ввода-вывода:

select/poll:

- книга “Роберт Лав — Linux. Системное программирование, 2-е издание, Питер, 2014” — глава 2. Файловый ввод-вывод, Раздел “Мультиплексный ввод-вывод”, стр. 81;
- книга “Стивенс У. Р. — UNIX: разработка сетевых приложений, 3-е изд., Питер, 2007” — глава 6. “Мультиплексирование ввода-вывода: функции select и poll”, стр. 185;
- книга “Стивенс Р. — UNIX. Профессиональное программирование, 2-е изд., Символ-Плюс, 2007” — глава 14. Расширенные операции ввода-вывода, раздел 14.5 Мультиплексирование ввода-вывода, стр. 558
- книга “Michael Kerrisk — The Linux Programming Interface, 2010” — глава 63. Alternative I/O Models, раздел 63.2 I/O Multiplexing, стр. 1374

epoll:

- книга “Роберт Лав — Linux. Системное программирование, 2-е издание, Питер, 2014” — глава 4. Расширенный файловый ввод-вывод, Раздел “Опрос событий”, стр. 131
- книга “Michael Kerrisk — The Linux Programming Interface, 2010” — глава 63. Alternative I/O Models, раздел 63.4 The epoll API, стр. 1399
- Epoll

kqueue:

- книга “Стивенс У. Р. — UNIX: разработка сетевых приложений, 3-е изд., Питер, 2007” — глава 14. “Дополнительные функции ввода-вывода”, раздел 14.9. Расширенный опрос, стр. 436
- kqueue tutorial
- ман-страница в интеренете
- События ядра в FreeBSD
- Краткое введение в kqueue/kevent

■ *Вячеслав Коваль*

3. Локальная установка и использование Perl-модулей

Рассмотрены способы установки Perl-модулей в локальную директории с помощью `cpanm`, использование `local::lib` для работы с ними, и `carton` для автоматизации процесса.

Часто требуется изолировать установленные модули для каждого проекта отдельно. Это может быть связано с разными версиями, необходимостью локализации или, например, с нежеланием устанавливать модули в систему, если вы просто тестируете какое-то новое Perl-приложение. Для примера возьмем простейшую серверную программу на `Plack`, которая при наличии параметра `name` приветствует посетителя:

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

use strict;
use warnings;

use Plack::Request;

sub {
    my $env = shift;

    my $req = Plack::Request->new($env);
    my $name = $req->param('name') || 'anonymous';

    return [200, [], [qq{Hello, $name!}]];
}
```

Наш проект на текущий момент имеет следующую структуру:

```
app.psgi
```

cpanm

Далее с помощью `cpanfile` (подробнее о формате `cpanfile` читайте в статьей Что такое cpanfile?) укажем зависимости:

```
requires 'Plack';
```

Теперь наш проект выглядит так:

```
app.psgi
cpanfile
```

Установим зависимости в локальную директорию. Пусть, например, это будет `third-party`. Для того, чтобы все модули устанавливались несмотря на их присутствие в локальной системе, воспользуемся ключом `-L` у `cpanm`:

```
$ cpanm -L third-party --installdeps .
```

Установка займет некоторое время и на выходе мы получим следующую структуру:

```
app.psgi
cpanfile
third-party/
    bin/
        ...
        plackup
    lib/
        perl5/
            x86_64-linux-gnu-thread-multi
            ...
    man/
        ...
```

Как видно, модули установились в `third-party/lib/perl5`, исполняемые файлы в `third-party/bin` и man-страницы в `third-party/man`. В директории `perl5` присутствует также `x86_64-linux-gnu-thread-multi`, куда обычно устанавливаются модули, требующие компиляции. К сожалению, до сих пор `Plack` требует компилятор для установки, однако ведутся работы по разделению дистрибутива на несколько пакетов, где необходимые модули для запуска приложения не будут требовать компиляции.

Работа `cpanm` на этом закончена. Мы установили необходимые модули в локальную директорию. Теперь необходимо запустить наше приложение.

local::lib

Если мы попробуем запустить следующим образом:

```
$ perl third-party/bin/plackup app.psgi
```

То получим вполне ожидаемую ошибку:

```
Can't locate Plack/Runner.pm in @INC
```

Мы, конечно, можем указать `perl` с помощью ключа `-I` где искать модули, однако это довольно утомительно. Более того, часто можно забыть подключить директорию с компилированными модулями, и это не так просто автоматизировать, потому как название директории меняется в зависимости от операционной системы. Именно для подключения локальных модулей нам поможет `local::lib`.

```
$ perl -Mlocal::lib=third-party third-party/bin/plackup
    app.psgi
HTTP::Server::PSGI: Accepting connections at http
    ://0:5000/
```

Ключ `-M` интерпретатора подключает модули до запуска скрипта. А с помощью `=` можно передать модулю параметры. `local::lib` получает их в методе `import` и подключает необходимую директорию.

Использование `cpanm` и `local::lib` таким образом позволяет быстро установить модули рядом с проектом и запустить его. Однако, для больших приложений, которые часто приходится устанавливать и запускать на разных системах, это не слишком удобно.

Carton

`Carton` объединяет в себе возможности `cpanm` и `local::lib` (на самом деле, внутри он использует эти два модуля), а также избавляет от необходимости ручного указания путей, позволяет контролировать зависимости и упрощает процесс поставки приложения.

Для работы `carton` достаточно `cpanfile`, который у нас уже есть. Вернемся к структуре проекта до установки зависимостей и иници-

ализируем carton:

```
$ carton install
Installing modules using cpanfile
Successfully installed ...
23 distributions installed
Complete! Modules were installed into local
```

Модули установились в директорию local, но это не так важно, потому как carton сам позаботится о добавлении нужных путей для запуска perl. Кроме того, carton создал файл cpanfile.snapshot, где указаны установленные зависимости с их версиями. Это файл стоит добавить в систему контроля версий. При запуске carton install --deployment и при наличии cpanfile.snapshot будут установлены указанные там версии. Таким образом, на машине другого разработчика или сервере будет такое же окружение.

Запускаем приложение:

```
$ carton exec -- plackup app.psgi
HTTP::Server::PSGI: Accepting connections at http
    ://0:5000/
```

carton exec запускает приложение в виртуальном окружении, где все модули подгружаются из директории local. Так, например, можно проверять все ли модули указаны как зависимости.

carton также позволяет собирать дистрибутив с зависимостями:

```
$ carton bundle
Bundling modules using cpanfile
Copying K/KA/KAZEBURO/Apache-LogFormat-Compiler-0.32.tar.
    gz
Copying ...
/tmp/h3lGuLzMVc/carton.pre.pl syntax OK
Bundling vendor/bin/carton
Complete! Modules were bundled into vendor/cache
```

Теперь в директории vendor/cache находятся тарболы модулей. Просто скопировав их на другую машину и запустив там:

```
$ carton install --cached
```

модули будут устанавливаться из тарболов без обращения к CPAN. Так можно разворачивать приложения на серверах без доступа к

внешним сетям.

Другие решения

Другим подходом для контроля зависимостей является создание своего приватного CPAN. Это позволяют сделать модули `CPAN::Mini` и подобные ему. Отдельно стоит приложение `Pinto` (читайте подробнее в статье Pinto — собственный CPAN из коробки), которое автоматизирует процесс установки, контроля и фиксирования модулей.

■ *Вячеслав Тихановский*

4. Работа с API GitHub в Perl

Github один из самых популярных сервисов для создания, распространения и совместной работы с программным обеспечением. GitHub обладает превосходным API, позволяя автоматизировать задачи администрирования и тестирования, а также создавать уникальные сервисы, приносящие радость разработчикам. И, как всегда, CPAN предлагает отличный выбор библиотек для работы с GitHub API в Perl.

GitHub API

Перед обзором средств для работы с GitHub API необходимо ознакомиться с тем, что оно из себя представляет. Любое обращение к API делается с помощью http-запросов к сервису api.github.com, поэтому начать изучение можно имея под рукой `curl` или аналогичную утилиту. Сервис возвращает ответ в формате JSON, причём немаловажное значение имеют и http-заголовки ответа. Например, информацию о пользователе можно получить, выполнив GET-запрос по URI `/users/:user`:

```
$ curl -i https://api.github.com/users/octocat

HTTP/1.1 200 OK
Server: GitHub.com
Date: Mon, 06 Oct 2014 06:18:13 GMT
Content-Type: application/json; charset=utf-8
X-RateLimit-Limit: 60
X-RateLimit-Remaining: 59
X-RateLimit-Reset: 1412579893
X-GitHub-Media-Type: github.v3

{
  "login": "octocat",
  "id": 583231,
  "avatar_url": "https://avatars.githubusercontent.com/u
     /583231?v=2",
  "gravatar_id": "",
  "url": "https://api.github.com/users/octocat",
  ...
}
```

Здесь для простоты удалены некоторые заголовки и часть ответа. Как и ожидалось, тип содержимого ответа — это `application/json`. Заголовки, начинающиеся с `X-` не относятся к спецификации HTTP и в данном случае несут информацию, относящуюся к API сервиса:

- `X-GitHub-Media-Type` сообщает, что используется третья версия API,
- `X-RateLimit-...` информирует о том, каков ваш лимит на обращения к данному API.

Итак, как уже удалось выяснить, GitHub API предоставляет возможность выполнять некоторые запросы анонимно, но ограничивает количество таких обращений до 60 за час. Для того, чтобы увеличить данный лимит, требуется выполнять авторизацию. Для авторизованного клиента лимит увеличивается до 5000 обращений за час. На данный момент GitHub предоставляет несколько способов авторизации:

- Basic-авторизация — каждый запрос сопровождается указанием имени зарегистрированного пользователя и пароля;
- OAuth-авторизация — запрос сопровождается указанием ключа доступа (*access token*), который даёт определённый набор прав.

Если первый вариант авторизации может быть приемлем на этапе знакомства с API, то для реального применения он мало походит ввиду своей небезопасности (нужно передавать пароль в приложение, которое получает максимальный набор прав), а также в случае использования двухфакторной авторизации, когда помимо пароля на каждую операцию требуется одноразовый код. Поэтому в основном применяется авторизация с использованием ключей доступа, и на их видах и способах получения надо остановиться подробно.

Ключи доступа

Ключи доступа позволяют авторизовать ваше приложение на выполнение ограниченного набора действий (*scopes*), как то чтение/-запись репозиториев, создание/удаление заметок gist и т.п. Ключи доступа можно отзывать в случае компрометации. Всё это делает их довольно безопасным средством для работы с API. Существуют два подхода к созданию ключей.

Персональные ключи доступа

Это самый простой способ получения ключа доступа для вашего приложения. С помощью веб-интерфейса настроек приложений создаётся ключ доступа и указываются те права, которые получает приложение. С использованием подобных ключей вы можете решать задачи по администрированию или тестированию ваших репозиториев, получения публичной информации о других репозиториях и пользователях.

Регистрация приложения

Регистрация приложения — это способ, который позволяет получать ключи доступа в соответствии со спецификацией OAuth2. Вы регистрируете приложение на GitHub. Пользователь GitHub, который хочет воспользоваться сервисом вашего приложения, предоставляет требуемые разрешения, и GitHub генерирует соответствующий ключ доступа для вашего приложения. В основном такой подход используется для веб-приложений, в случае если требуется аутентифицировать GitHub-пользователя или приложение должно действовать от имени пользователя (как например, Travis-CI).

Модули GitHub API на CPAN

На данный момент наиболее популярны два модуля для работы с GitHub API: Pithub и Net::GitHub. Оба модуля обладают практически полным набором функций и похожи как братья близнецы, можно отметить лишь два заметных отличия:

- `Pithub` больше ориентирован на ООП-подход в программировании. Например, результат обращения к API — это объект класса `Pithub::Result`, который, в частности, поддерживает метод `count`, чтобы получить число элементов, и метод `next`, чтобы получить следующий элемент. В то время как `Net::GitHub` возвращает голый результат, например, в виде хеша.
- `Net::GitHub`, наряду с ключами доступа, поддерживает и basic-авторизацию, что может быть полезно, чтобы сгенерировать новый ключ доступа. `Pithub` работает только с ключами доступа.

Возможно, это дело вкуса, но `Pithub` показался мне более удобным и приятным в использовании, поэтому остановимся подробно именно на нём.

Структура модулей Pithub

API GitHub можно разделить на несколько основных разделов, каждый из которых представляется соответствующим модулем:

- `Pithub::Events` — API событий, позволяющая отслеживать ту или иную активность на сайте GitHub;
- `Pithub::Gists` — API выжимок (*gist*) для получения и манипуляций с выжимками;
- `Pithub::GitData` — API данных для работы с сущностями git-репозиториев: блобы, деревья, коммиты, теги и ссылки;

- `Pithub::Issues` — API баг-трекера (ошибки, вопросы и проблемы);
- `Pithub::Orgs` — API организаций для получения и изменения информации организаций, а также управления командами и участниками;
- `Pithub::PullRequests` — API запросов на слияние, включая и работу с комментариями к запросам;
- `Pithub::Repos` — API репозиториев, содержащее обширное число операций по работе с репозиториями;
- `Pithub::Users` — API пользователей для получения и изменения информации о пользователях GitHub.
- `Pithub::Search` — API поиска по GitHub.

Как видно из списка, в `Pithub` пока отсутствуют некоторые новые разделы GitHub API, такие как *Активность*, включающая несколько подразделов: уведомлений, ATOM-фидов и других. Также отсутствуют *Markdown API*, *Meta API*, *RateLimit API* и некоторые другие. Кроме того `Pithub::Search` использует устаревший *legacy*-интерфейс поиска. Тем не менее, как будет показано позже, с помощью модуля `Pithub::Base` вы можете компенсировать отсутствие готового нужного раздела API и напрямую делать вызовы GitHub API.

“Hello, GitHub!”

Итак, вы успешно зарегистрировали нового пользователя на GitHub (увы, некоторые просчёты могут привести к блокировке учётной записи, так что лучше перестраховаться) и создали первый персональный ключ доступа. Можно приступать к созданию своего первого приложения.

```
1 use DDP;
2 use Pithub;
3
4 # Ключ доступа берем из переменной окружения
```

```
my $p = Pithub->new(
    token => $ENV{GITHUB_TOKEN}
);

# Исчерпывающая информация о пользователе octocat
my $result = $p->users->get( user => 'octocat' );

# Вывод декодированной структуры ответа
p $result->content;

# Вывод количества оставшихся запросов к API
p $result->ratelimit_remaining;
```

Перед запуском скрипта необходимо экспортировать переменную окружения `GITHUB_TOKEN`, в которую нужно поместить значение ключа доступа:

```
$ export GITHUB_TOKEN=deadbeeffacecafe
$ perl hello_github.pl

\ {
    avatar_url  "https://avatars.githubusercontent.com/u
       /583231?v=2",
    bio         undef,
    blog        "http://www.github.com/blog",
    company     "GitHub",
    created_at  "2011-01-25T18:44:36Z",
    email       "octocat@github.com",
    ...
}
4999
```

В данном примере мы создавали базовый объект `Pithub` `$p`, а затем получали доступ к методам объекта `Pithub::Users` с помощью метода `users`. Точно такой же результат можно было получить, создав `Pithub::Users` напрямую:

```
my $u = Pithub::Users->new(
    token => $ENV{GITHUB_TOKEN}
);

my $result = $u->get( user => 'octocat' );
```

Такой же подход можно использовать и для всех других субмодулей `Pithub`.

Работа со списками

Некоторые вызовы API возвращают не один объект, а список JSON-объектов. Как, например, запрос списка репозиториев пользователя или организации:

```
use DDP;
use Pithub;

my $r = Pithub::Repos->new( token => $ENV{GITHUB_TOKEN} )
    ;

my $result = $r->list( org => 'PadreIDE' );

p $result->count;

while ( my $row = $result->next ) {
    p $row->{name};
}
```

В данном примере запрашивается список репозиториев организации *PadreIDE*. Запрос возвращает список из 30 репозиториев, но самих репозиториев на самом деле 100. Связано это с тем, что GitHub по умолчанию устанавливает лимит на 30 объектов в рамках одного запроса, и для того, чтобы получить весь список объектов, можно запрашивать их постранично:

```
my $result = $r->list( org => 'PadreIDE' );
while ( $result && $result->success ) {
    while ( my $row = $result->next ) {
        p $row->{name};
    }
    $result = $result->next_page;
}
```

В данном случае метод `next_page` делает новый запрос к API и возвращает новую порцию данных (или `undef`, если данных больше нет).

Тоже самое можно получить, установив флаг `auto_pagination` в истинное значение, тогда метод `next` будет автоматически вызывать `next_page` при завершении списка. Кроме того, можно регулировать и количество объектов в одном ответе с помощью свойства `per_page`:

```
# включить автоматический запрос последующих страниц
$r->auto_pagination(1);

# 50 объектов на странице
$r->per_page(50);

my $result = $r->list( org => 'PadreIDE' );
while ( my $row = $result->next ) {
    p $row->{name};
}
```

Запросы к API, которые не реализованы в `Pithub`

GitHub API активно развивается, добавляются новые разделы, которые не были реализованы в `Pithub`. Поскольку все обращения к API выполняются через http-запросы, в `Pithub` существует возможность указывать нужный метод и URI запроса с помощью метода `request`. Рассмотрим на примере *Ratelimit API*, которое позволяет GET-запросом по URI `/rate_limit` получить информацию об актуальных значениях лимитов на API-запросы для вашего приложения.

```
use DDP;
use Pithub;

my $p = Pithub->new(
    token => $ENV{GITHUB_TOKEN}
);

my $result = $p->request(
    method => 'GET',
    path   => '/rate_limit',
);
p $result->content;
```

В данном примере мы указываем метод и путь запроса к API, и в результате получаем объект класса `Pithub::Result` с информацией.

```
\ {
    rate        {
        limit       5000,
        remaining   5000,
        reset       1412552655
```

```
    },
    resources   {
        core     {
            limit       5000,
            remaining   5000,
            reset       1412552655
        },
        search   {
            limit       30,
            remaining   30,
            reset       1412549115
        }
    }
}
```

Другой пример это *Markdown API*. GitHub позволяет преобразовывать данные, отправленные в формате Markdown, в html-формат.

```
use DDP;
use Pithub;

my $p = Pithub->new(
    token => $ENV{GITHUB_TOKEN}
);

my $result = $p->request(
    method => 'POST',
    path   => '/markdown',
    data => {
        text => '**Hello, World!**',
        mode => 'markdown',
    }
);

p $result->raw_content;
```

В данном примере выполняется POST-запрос, тело запроса в формате JSON формируется из структуры `data`. Обратите внимание, что для извлечения результата используется метод `raw_content`, поскольку получаемые данные приходят в формате `text/html`, а не `application/json` и не требуют декодирования.

```
<p><strong>Hello, World!</strong></p>
```

Практический пример: строим свой CI-сервер

Одно из интересных применений GitHub API — это возможность создания своих собственных сервисов и в частности сервера непрерывной интеграции.

В GitHub существует возможность уведомлять внешние сервисы о тех или иных событиях, происходящих в ваших репозиториях. Например, отправка коммитов в репозиторий (push). В настройках каждого репозитория в разделе *WebHooks* можно указать URL сервиса, который будет принимать POST-запросы с информацией о произошедшем событии.

API GitHub поддерживает установку статусов для каждого коммита. Статус — это некоторая метка, информация о состоянии репозитория и может использоваться CI-системами для обозначения того, успешно ли собирается проект или того, что данный коммит приводит к ошибкам.

Рассмотрим простое веб-приложение на веб-фреймворке `Dancer`, которое может выступить в роли CI-сервера:

```
use Dancer;
use Pithub;
use JSON ();

post '/event_handler' => sub {
    my $payload = JSON::decode_json(request->body);

    my $user = $payload->{repository}->{owner}->{name};
    my $repo = $payload->{repository}->{name};

    # Статус сборки
    my $s = Pithub::Repos::Statuses->new(
        token => $ENV{GITHUB_TOKEN},
        user  => $user,
        repo  => $repo,
    );

    # Статус сборки — в обработке (pending)
    $s->create(
        sha => $payload->{after},
        data => {
            state => 'pending',
```

```
23              description => 'starting build',
24              context => 'ci/perl',
25          }
26      );
27
28      # Запустить сборку
29      run_ci(
30          commit => $payload->{after},
31          url    => $payload->{repository}->{clone_url},
32          name   => $payload->{repository}->{name},
33          user   => $payload->{repository}->{owner}->{name
                },
34          cb     => sub {
35              my ( $state, $url ) = @_;
36
37              # Обновляем статус (failure или success)
38              $s->create(
39                  sha => $payload->{after},
40                  data => {
41                      state => $state,
42                      description => "build $state!",
43                      context => 'ci/perl',
44                      target_url => $url
45                  }
46              );
47          },
48      );
49
50      "ok";
51  };
52
53  dance;
```

Итак, запускается веб-сервер, который ожидает POST-запрос по URI `/event_handler`, которое было указано в *WebHooks* репозитория. GitHub на каждый push в репозиторий будет отправлять данные на наше веб-приложение. Данные приходят в формате JSON, которые декодируются в строке 6. Из полученной структуры извлекается информация о коммите, пользователе и названии репозитория.

В строке 19 мы используем API статусов и устанавливаем для полученного коммита статус «pending», т.е. коммит находится в процессе проверки. Далее следует некая асинхронная процедура `run_ci`, которая клонирует репозиторий, выполняет проверку, например, сборку. И по результатам выполняет функцию обратного

вызова с результатами проверки: это может быть «failure»/«error» или «success». Этот результат мы используем для обновления статуса коммита. В данном примере мы также устанавливаем `target_url` — это url по которому можно получить, например, логи сборки репозитория.

В веб-интерфейсе GitHub информацию о статусе можно увидеть в списке веток репозитория, где показывается последний статус для каждой ветки в виде ссылки, которую вы указали в `target_url`.

Таким же образом можно проверять сборку репозитория при получении запросов на слияние (pull request). В этом случае надо проверять получаемый заголовок `X-Github-Event`, который будет иметь значение `pull_request`.

Заключение

GitHub API — это уникальный сервис, открывающий поистине огромные возможности для создания полезных и практичных приложений. В статье было рассмотрено лишь несколько примеров по работе с API с помощью модуля `Pithub`, которые дают представление о способе его работы и интерфейсе. Более подробная информация содержится в POD-документации модуля, а информация об API GitHub — на сайте для разработчиков. Смотрите и изучайте, надеюсь, его больше никогда не заблокируют.

■ *Владимир Леттиев*

5. Введение в Rose::DB::Object

Рассмотрено использование простейших возможностей ORM-пакета `Rose::DB::Object`

Автор пишет, что `Rose::DB::Object` — “object-relational mapper (ORM)”. Это на 50% верно. Мы можем получить доступ к базе, но не можем создать объект Perl и создать в базе структуру под него. То есть это удобная замена SQL-запросам, не более.

Что нам даст `Rose::DB::Object`

Мы получаем возможность простые операции делать ещё проще, но при этом сложные операции становятся очень запутанными и нетривиальными. Реально можно использовать выборку с не очень сложными условиями, создание (INSERT) — удобно, обновление (UPDATE) — очень удобно, удаление (DELETE) — не сильно удобнее чем с `DBI`.

Удобно использовать `Rose::DB::Object` в операциях, затрагивающих один объект. В этом случае задействуется вся мощь внешних ключей, автоматического преобразования дат в объекты `DateTime`, обнаружение некоторых неправильно передаваемых данных ещё до обращения к базе данных (используется ранее созданная модель базы). Использовать в операциях с большим числом объектов уже надо очень осторожно. При этом `Rose::DB::Object`, по сообщениям автора, в 20 раз производительней единственного прямого конкурента — `DBIx::Class`. Мой опыт — надо совмещать чистый SQL и Rose.

Текст ниже описывает основные операции. Для того, чтобы протестировать все примеры, обращаётесь к репозиторию https://github.com/Pilat66/RoseDBObject. Примеры из этой статьи находятся в файле example_1.pl

Структура тестовой базы данных

База данных у нас будет из двух таблиц.

```
my $sql_create = <<END;

CREATE TABLE authors (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT
);

CREATE TABLE books (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT,
    editor_id INTEGER,
    author_id_1 INTEGER,
    author_id_2 INTEGER,
    FOREIGN KEY (editor_id) REFERENCES authors (id) ,
    FOREIGN KEY (author_id_2) REFERENCES authors (id) ,
    FOREIGN KEY (author_id_1) REFERENCES authors (id)
);

END
```

В таблице `books` нетривиальный момент то, что есть два почти одинаковых столбца `author_id_1` и `author_id_2` — они нужны для демонстрации дописывания генератора модулей в разделе 'Создание классов `Rose` и `ORM::ConventionManager`.

В стиле `DBI`

Сначала короткий пример работы с нашей базы стандартными средствами. Я создаю тестовую базу данных и заполняю её тестовыми данными. Этот пример занимает одну страницу текста, следующий пример выполняет то же самое в стиле `Rose::DB::Object`, но занимает в несколько раз больше строк текста :)

```
my $dbh = DBI->connect("dbi:SQLite:dbname=$dbname_dbi", "
    ", "");
$dbh->{AutoCommit} = 0;
$dbh->do('PRAGMA foreign_keys = ON');

foreach (1 .. 3) {
```

```
6   $dbh->do("INSERT INTO authors(name) VALUES(?)", {}, "
       Author $_");
7 }
8
9 foreach (1 .. 3) {
10  $dbh->do( <<END, {}, "Book $_", 1, $_);
11  INSERT INTO books(name, author_id_1, author_id_2)
12  VALUES(
13      ?,
14      (SELECT id FROM authors WHERE name like '%'||?),
15      (SELECT id FROM authors WHERE name like '%'||?)
16  )
17 END
18 }
19
20 $dbh->commit;
```

В стиле `Rose::DB::Object`

В отличии от предыдущего примера, перед работой с ORM `Rose::DB::Object` надо провести дополнительную работу:

- подключиться к базе;
- сформировать несколько служебных классов, корректирующих поведение ORM;
- превратить существующую структуру базы данных в объектную модель;
- загрузить эту модель.

Довольно сложный комплекс действий. Собственно использование ORM начинается с раздела “Основные операции с базой”.

Главное, что надо сразу понять о `Rose::DB::Object` — он (или оно) построен на системе статических классов (модулей Perl). Каждой схеме в базе соответствует (в один момент времени свой набор модулей, который нельзя использовать для манипуляции другой такой же схемой или обращаться к той же схеме с другим именем пользователя.

Связано это с тем, что ради простоты использования, каждый модуль (отражающий структуру таблицы базы) имеет свой `$dbh` для доступа к базе (точнее `$dbh` прячется в объекте `$db`, унаследованном от `Rose::DB`), причём этот `$dbh` используется неявно в конструкциях типа:

```
1 $p = Product->new(name => 'Sled');
2 $p->save;
```

Это конструкция создания новой строки в базе. Мы нигде не указываем о какой базе идёт речь — это указывается во время инициализации системы классов. Правда, указать всё же можно, но тогда нарушается гармония:

```
1 $p = Product->new(db => $db, name => 'Sled');
```

так как этот `$db` ещё надо как-то хранить, создавать, и вообще заботиться о нём. Потом об этом напишу подробнее.

Второе, что надо знать. Я пишу исключительно с использованием `Rose::DB::Object::Loader` — то есть сначала создаём базу данных, потом `Loader` строит множество модулей, потом мы их инициализируем и используем. Теоретически можно модули (классы) создавать и самостоятельно; но для больших баз, а с маленькими Rose нет смысла использовать, это слишком сложно. Проще сделать скрипт, создающий дерево классов, и выполнять его при каждом изменении базы.

Время от времени задаются в форумах вопросы типа “можно ли обратить процесс, то есть создавать структуру базы данных по `Rose`-модели”. Ответ на это — можно, так как модель имеет всё необходимое для этого, но `Rose` это не делает. И делать это не нужно, всё таки ORM в Perl это совсем не то же что Hibernate в Java.

Настройка `Rose::DB::Object`

Дополнительные классы

Дополнительные классы позволят изменить метод подключения к базе данных и правила автосоздания классов Rose по существующей базе данных.

`ORM::DB`, `ORM::DB_Object`, `ORM::ConventionManager` в настоящем проекте надо описывать в отдельных файлах. Я их сделаю прямо тут. ORM — это префикс дерева классов, его можно выбрать любым или вообще не использовать, но тогда может возникнуть путаница.

`ORM::DB` `ORM::DB` содержит описание подключения к нашей базе.

В данном случае это псевдоподключение, так как будет использоваться метод, отличный от стандартного. То, что необходимо — сообщить `Rose` тип базы данных:

```
1 package ORM::DB;
2 use strict;
3 use base qw(Rose::DB);
4
5 __PACKAGE__->use_private_registry;
6 __PACKAGE__->register_db(driver => 'sqlite',);
7
8 1;
```

`ORM::DB_Object` `ORM::DB_Object` — это тот класс, который rose будет использовать вместо `Rose::DB::Object`. Нам нужно только переопределить метод `init_db()`.

```
1 package ORM::DB_Object;
2
3 use strict;
4 use base qw(Rose::DB::Object);
5
6 our $DB;
7
8 sub init_db {
```

```
  my $self = shift;
  return $DB;
}

1;
```

ORM::ConventionManager ORM::ConventionManager переопределяет некоторые алгоритмы создания классов Rose::DB по существующей структуре базы данных с помощью Rose::DB::Object::Loader .

auto_foreign_key_name() позволяет изменить стандартные имена для ссылки на внешние таблицы, table_to_class() позволяет определить имена классов, соответствующих таблицам в модели. Например, для таблицы authors можно выбрать класс ORM::Authors , ORM::Author, ORM::authors, ORM::author или любой другой. По-умолчанию, Rose::DB::Object::Loader применяет хитрые алгоритмы создания имён, настолько хитрые, что это приносит больше вреда чем пользы и создаёт проблемы. Для создания имён есть встроенные флаги, но проще всего иметь имена классов, совпадающих с именами таблиц в нижнем регистре. Но это всё на любителя. Но на практике имена таблиц бывают очень разные.

```
package ORM::ConventionManager;
use base qw(Rose::DB::Object::ConventionManager);

sub auto_foreign_key_name {
  my ($self, $f_class, $current_name, $key_columns,
     $used_names) = @_;
  my $name;

  if ($f_class eq 'ORM::authors') {
    $name = 'author_primary'   if ($key_columns->{
       author_id_1});
    $name = 'author_secondary' if ($key_columns->{
       author_id_2});
  }
  else {
    $name = Rose::DB::Object::ConventionManager::
       auto_foreign_key_name(@_);
  }
  return $name;
}

```

```
sub table_to_class {
  my ($self, $table, $prefix, $plural) = @_;
  $table = lc $table;
  return ($prefix || '') . $table;
}

1;
```

Подключение к базе данных, создание объекта `Rose::DB`

```
my $dbh = DBI->connect("dbi:SQLite:dbname=$dbname_rose",
    "", "");
$dbh->{AutoCommit} = 0;
```

`private_pid` и `private_tid` `Rose::DB` проверяет параметры `private_pid` и `private_tid`, поэтому надо их создать.

```
$dbh->{'private_pid'} = $$;
$dbh->{'private_tid'} = threads->tid if ($INC{'threads.pm
    '});
```

`$schema` `$schema` содержит имя схемы базы данных.

У нас может быть несколько схем с одинаковой структурой, и мы можем указать, с какой схемой нужно работать. `$db->schema($schema)` можно вызывать в любой момент. SQLite3 не имеет схем, Так что в этом примере `$schema=undef`.

```
my $schema = undef;

my $db = ORM::DB->new();
$ORM::DB_Object::DB = $db;

$db->dbh($dbh);
$db->schema($schema) if $schema;

my $module_dir = './ormlib_auto';
```

В директории ormlib_auto будут лежать сгенерированные модули описания базы данных. Перед созданием модулей надо либо очистить эту директорию, либо обеспечить её отсутствие в @INC.

```
#system("rm -Rf $module_dir"); # надо сделать перед созданием модулей
system("mkdir -p $module_dir");

my $loader = Rose::DB::Object::Loader->new(
  db                 => $db,
  db_schema          => $schema,
  module_dir         => $module_dir,
  class_prefix       => 'ORM',
  db_class           => 'ORM::DB',
  base_classes       => ['ORM::DB_Object'],
  convention_manager => 'ORM::ConventionManager',

# include_views можно и установить в 1, но будут глюки.
# Да и Rose любит иметь приватные ключи, роль которых будет исполнять
# столбец из view, выбранный случайным образом.
  include_views => 0,

# include_tables можно не указать, тогда будут использоваться все таблицы
# ( если require_primary_key==1, то только те таблицы,
# у которых есть primary key)
  include_tables => [qw{ books authors }],

# exclude_tables содержит список таблиц, для которых
# не нужно создавать классы
  exclude_tables => [],

# require_primary_key позволяет указать, создавать ли классы
# для таблиц без private key. Таблицы без primary key порождают глюки,
# поэтому лучше их не загружать. Ну или аккуратно следить за тем,
# чтобы их использовать только в ::Manager-> запросах.
  require_primary_key => 1,

# warn_on_missing_pk порождает сообщения о потенциальных проблемных таблицах
#warn_on_missing_pk  => 1,

);
```

Создание классов `Rose` Классы создать можно двумя способами: вызвать `$loader->make_modules` или `$loader->make_classes()`. Первый метод создаст файлы с модулями Perl в указанной module_dir директории, второй только создаст и загрузит сами классы.

Есть соблазн всегда пользоваться только `make_classes()` и не замусоривать файловую систему. Делать это не надо по двум причинам:

1. изучение созданных модулей крайне полезно для отладки;
2. создание классов на базах данных с большим числом таблиц крайне ресурсоёмкое занятие. Создание классов может занимать 20 секунд, а их загрузка из файлов одну секунду.

Дополнительно о фрагменте:

```
post_init_hook => sub {
  my $meta = shift;
  $meta->{allow_inline_column_values} = 1;
},
```

Очень хорошая особенность `Rose` — он позволяет вмешиваться в работу его компонентов на разных стадиях, например в данном случае после создания метаинформации для генерации объекта, соответствующего таблице, можно эту метаинформацию немного подправить. Конкретно `allow_inline_column_values` в данном случае не нужно ни для чего, это только демонстрация. Вообще очень полезная опция, Rose может сам установить значения по-умолчанию для некоторых столбцов, для которых в баые указан параметр `default`, а может позволить выполнить это самой СУБД. Причина простая — некоторые `default` значения в базе на самом деле не константы, а функции (`now()` в PostgreSQL, SYSDATE в Oracle), и Rose далеко не всегда справляется сам с опознанием таких объектов.

`Rose::DB::Object::Loader` имеет кучу флагов, рекомендую почитать о них. Так же полезно посмотреть исходники — они небольшие по размеру, и позволяют понять некоторые нетривиальные вещи.

```
my @classes = $loader->make_modules(
```

```
  db             => $db,
  post_init_hook => sub {
    my $meta = shift;
    $meta->{allow_inline_column_values} = 1;
  },
);

ddx @classes;

#   "ORM::authors",
#   "ORM::authors::Manager",
#   "ORM::books",
#   "ORM::books::Manager",

push @INC, 'ormlib_auto';
```

Подключение классов `ORM::` В данном примере загружать классы не нужно, так как `Loader` их уже загрузил. Но в реальности `Loader` запускается один раз и сохраняет классы в файлах, потом классы только загружаются без вызова `Loader`.

Ещё надо не забывать загружать `ORM::DB`, `ORM::DB_Object`, `ORM::ConventionManager`. Мы их уже загрузили, да и вообще их нет в файловой системе, так что это не делаем, но помним что в реальном проекте сделать надо, например

```
foreach(@classes){
    load($_);
}

foreach (1 .. 3) {
  my $author = ORM::authors->new(name => "Author $_");
  $author->save();

}

ddx $dbh->selectall_arrayref("SELECT * FROM authors", {
   Slice => {}});

#   { id => 1, name => "Author 1" },
#   { id => 2, name => "Author 2" },
#   { id => 3, name => "Author 3" },

#my $authors = ORM::authors::Manager->get_authors();
#ddx $authors;
```

```
19
20 foreach (1 .. 3) {
21   ORM::books->new(
22     name             => "Book $_",
23     editor           => ORM::authors->new(id => 1)->load,
24     author_primary   => ORM::authors->new(id => 1)->load,
25     author_secondary => ORM::authors->new(id => $_)->load
         ,
26   )->save();
27 }
28
29 ddx $dbh->selectall_arrayref("SELECT * FROM books", {
     Slice => {}});
30
31 #   { author_id_1 => 1, author_id_2 => 1, id => 1, name
     => "Book 1" },
32 #   { author_id_1 => 1, author_id_2 => 2, id => 2, name
     => "Book 2" },
33 #   { author_id_1 => 1, author_id_2 => 3, id => 3, name
     => "Book 3" },
```

Основные операции с базой

Теперь быстренько пройдёмся по `Rose::DB::Object::Tutorial`. У нас есть таблицы `authors` и `books`. Над ними и поработаем.

Select — выборка данных

Загрузим автора, которого только что создали, и проверим, есть ли такой. `ORM::author->new(id => 1)` только создаёт объект в памяти, чтобы наполнить его данными из базы, надо вызвать функцию `load()`. Чтобы записать в базу — `save()`. Если объекта в базе нет, вызывается исключение. Параметр `speculative => 1` как раз предотвращает это.

```
1 my $author = ORM::authors->new(id => 1);
2 unless ($author->load(speculative=>1)) {
3   warn("Нет такого автора");
4 }
```

Но это загрузка только одного объекта, по его уникальному идентификатору. Можно загрузить несколько объектов, выполнив для них SQL-запрос. Подробно язык запросов описан в `Rose::DB::Object::Manager`.

Загрузка нескольких объектов выполняется двумя способами — аналоги функций DBI `$dbh->selectall_arrayref()` и `$sth->fetchrow_hashref()`. Первый способ подходит для небольших выборок, но он должен быть существенно быстрее второго, второй не загружает сразу все объекты в память, поэтому позволяет обработать любые объёмы информации. Сначала первый способ:

```
my $authors = ORM::authors::Manager->get_authors(
  query => [
    or => [
            name => {
                'like' => '%1%'
            },
            name => {
                'like' => '%2%'
           },
    ],
  ],
  select => ['name'],
  limit  => 2,
  offset => 0,
  debug  => 1
);
```

На экран выводится текст запроса, который пойдёт в базу, и подставляемые в него параметры:

```
SELECT
t1.name
FROM
  authors t1
WHERE
  (
    t1.name LIKE ? OR
    t1.name LIKE ?
  )
LIMIT 2 OFFSET 0 (%1%, %2%)
```

Опции:

- `debug => 1` заставит Rose вывести на консоль текст сформированного запроса;
- `limit => 1` — вывести одну строку, начиная с offset строк;
- `offset => 0` — пропустить ноль строк;
- `select => ['name']` — выбрать только столбец `name`, если это-то параметр не указывать, выбираются все столбцы.

В `$authors` будет содержаться массив объектов, соответствующих выбранным строкам таблицы.

Теперь второй способ, создадим итератор: разница только в функции `get_authors_iterator()` для выборки объектов.

```
my $authors = ORM::authors::Manager->get_authors_iterator
    (
  query => [or => [name => {'like' => '%1%'}, name => {'
     like' => '%2%'},],],
  select => ['name'],
  limit  => 2,
  offset => 0,
  debug  => 1
);

while (my $author = $authors->next) {
  ddx [$author->id, $author->name];
}
ddx "Total row(s):" => $authors->total;
```

Вне зависимости от метода, которым мы выбираем записи, мы получаем объекты, унаследованные от `Rose::DB::Object` — один объект, или массив объектов.

`get_authors` и `get_authors_iterator` — функции, их имена определяются в ConventionManager классе, и схему их образования можно изменить.

Получение и изменение значений столбцов

Для каждого столбца таблицы создаётся функция (`getter/setter`) с именем, по умолчанию совпадающим с именем столбца (можно

изменить в ConventionManager).

Не забываем, что мало вызвать $author->name('Author 1 New Name'), надо после этого сделать $author->save() и, возможно, $dbh->commit() (в нашем случае точно надо, так как базу мы открывали с опцией AutoCommit => 0).

```
my $author = ORM::authors->new(id => 1);
$author->load();
$author->name('Author 1 New Name');
$author->save();
```

Insert — добавление (создание) данных

```
my $new_author = ORM::authors->new(name => "Author 4");
$new_author->save();
```

Разница между получением данных и обновлением

Несмотря на то, что в обоих случаях используется метод new(), при получении данных мы должны указать уникальный идентификатор строки — первичный ключ или значение поля с уникальным индексом, после чего вызвать метод load(). Для создания данных первичный ключ можно не указывать, нужно указать значения для обязательных (not null) столбцов и вызвать метод save().

ОЧЕНЬ ВАЖНЫЙ МОМЕНТ!

При создании строк в таблице надо как-то задать значение для PRIMARY KEY столбцов. Можно сделать это в явном виде? указав значение столбца. Rose может сделать это автоматически, при соблюдении некоторых условий. Например, для PostgreSQL первичные ключи задаются с типом serial, который СУБД разворачивает в тип Integer (или BigInt), и создаёт последовательность с именем в стандартном формате. У MySQL есть встроенный тип AUTO_INCREMENT. У Oracle нет ничего, поэтому Rose предполагает, что для первичных ключей есть последовательность с именем:

```
my $name = join('_', $table, $column, 'seq');
```

Имя последовательности определяется в `ConventionManager`. Если Ваши имена последовательностей строятся иначе, можно в `ORM::ConventionManager` переназначить функции `auto_primary_key_column_sequence()` и `auto_column_sequence_name()` из `Rose::DB::Object::ConventionManager`.

Update — Обновление

Обновление так же просто — загружаем, обновляем, сохраняем.

```
my $author = ORM::authors->new(id => 1);
$author->load();
$author->name('Author 1 New New Name');
$author->save();
$author->load();
```

Если что-то хотим сделать потом с объектом, иногда надо повторно прочитать его из базы — при сохранении он мог быть изменён триггером, или ещё по каким-то причинам измениться.

Если надо обновить несколько объектов, используем подкласс `::Manager`:

```
my $num_rows_updated = ORM::authors::Manager->
    update_authors(
  set   => {
    name => {
        sql => "name || ' update'"
    },
  },
  where => [id => 1],
  debug => 1
);
```

Delete — Удаление

Удалять можно один объект или группу объектов. Сначала удалим один объект:

```
my $author = ORM::authors->new(id => 4);
$author->delete();
```

Теперь удалим несколько объектов.

```
my $num_rows_deleted
  = ORM::authors::Manager->delete_authors(where => [id =>
      {ge => '5'}],);
```

Более сложные операции — задействуем связи между объектами

Предположим, что наши книги имеют по паре авторов и одного редактора editor.

Таблица authors:

```
{ authorid1 => 1, authorid2 => 1, id => 1, name => "Book
    1" },
{ authorid1 => 1, authorid2 => 2, id => 2, name => "Book
    2" },
{ authorid1 => 1, authorid2 => 3, id => 3, name => "Book
    3" },
```

Таблица books

```
{ author_id_1 => 1, author_id_2 => 1, editor_id => 1, id
    => 1, name => "Book 1" },
{ author_id_1 => 1, author_id_2 => 2, editor_id => 1, id
    => 2, name => "Book 2" },
{ author_id_1 => 1, author_id_2 => 3, editor_id => 1, id
    => 3, name => "Book 3" },
```

Загрузим книгу:

```
my $book = ORM::books->new(id => 2)->load;
```

Теперь мы можем получить данные авторов, используя внешние ключи — просто вызвав:

```
$book->author_primary->name;
```

и

```
$book->author_secondary->name
```

Мы можем выбирать объекты, на которые ссылаемся, автоматически, по мере потребности. Побочный эффект от этого — дополнительные SQL запросы, то есть злоупотреблять этим нельзя. Правда, есть кэширование, но оно не панацея. Установим `Rose::DB::Object::Debug` в 1 и посмотрим как это выглядит:

```
$Rose::DB::Object::Debug = 1;
my $book = ORM::books->new(id => 2)->load;

    #SELECT author_id_2, editor_id, author_id_1, name, id
    #FROM books WHERE id = ? - bind params: 2

print $book->editor->name;

    #SELECT name, id FROM authors WHERE id = ? - bind
       params: 1

$Rose::DB::Object::Debug = 0;
```

Мы видим, что создался дополнительный SQL запрос. Затратно для большого количества книг, но при выводе информации об одной книге очень удобно.

Заключение

Рассмотрены базовые возможности. Вообще не описаны операции с связями типа many-to-many (многие-ко-многим), так как они для практического использования сложноваты. Не рассмотрены возможности выборок с связанными таблицами (имитация JOIN) — это тема для отдельной статьи. Например, из документации:

```
$products =
  Product::Manager->get_products(
    query =>
    [
      name => { like => 'Kite%' },
      id   => { gt => 15 },
    ],
    require_objects => [ 'vendor' ],
    with_objects    => [ 'colors' ],
    sort_by => 'name');
```

Получим SQL запрос

```
SELECT
      t1.id,
      t1.name,
      t1.vendor_id,
      t3.code,
      t3.name,
      t4.id,
      t4.name,
      t4.region_id
    FROM
      products t1
      JOIN vendors t4 ON (t1.vendor_id = t4.id)
      LEFT OUTER JOIN product_colors t2 ON (t2.product_id
          = t1.id)
      LEFT OUTER JOIN colors t3 ON (t2.color_code = t3.
         code)
    WHERE
      t1.id > 15 AND
      t1.name LIKE 'Kite%'
```

Мне кажется, что такого рода запросы проще делать руками. Но, используя знания о внешних ключах и связях между таблицами, такие запросы строятся просто. Сложно потом искать ошибки.

Сразу о замеченных тёмных местах в `Rose`

Огромная проблема — таблицы и столбцы с именами в смешанном регистре. Таблицу

```
CREATE TABLE "SmartTablE" ("Id": serial PRIMARY KEY, "
    NaMe": character varying);
```

использовать практически (почти) невозможно в существующей реализации.

Поля и таблицы, совпадающие с именами внутренних переменных Rose, придётся переименовать в `ConventionManager`.

■ *Константин Токар*

6. Обзор CPAN за сентябрь 2014 г.

Рубрика с обзором интересных новинок CPAN за прошедший месяц.

Статистика

- Новых дистрибутивов — 237
- Новых выпусков — 1044

Новые модули

- Clownfish

Проект *Clownfish* разрабатывается в рамках поискового движка Lucy, и представляет собой реализацию объектной системы для языка C. Одноимённый Perl-модуль Clownfish является *симбиотическим* байндингом, связывая объектную системы языка Perl с Clownfish, позволяя получать доступ к объектам Clownfish как к обычным объектам Perl, наследовать, расширять и затем использовать в Clownfish.

- App::MultiSsh

Утилита `mssh` позволяет выполнять команды через ssh-соединения одновременно (или последовательно) на нескольких хостах.

- SQL::QueryBuilder::Flex

Ещё один представитель модулей для генерации SQL-запросов. Интерфейс модуля оправдывает своё название, позволяя достаточно гибко строить весьма сложные запросы. Документация модуля содержит большое число примеров.

- Scalar::Watcher

Модуль `Scalar::Watcher` позволяет отслеживать изменение скалярной переменной, вызывая указанную пользователем функцию:

```
1    my $a = 123;
2    when_modified $a, sub { print "catch $_[0]\n" };
3    $a = 456; # напечатает 'catch 456'
```

- Rex::JobControl

Фреймворк для автоматизации задач администрирования `Rex` обзавёлся веб-интерфейсом `Rex::JobControl`. Веб-интерфейс создан на основе фреймворка *Mojolicious* и сервера задач Minion.

- Object::Util

`Object::Util` — набор полезных функций для работы с объектами. Идея интерфейса заимствована из `Safe::Isa`, но набор функций значительно шире и интереснее.

```
1    $foo->$_isa("Class");
```

В данном примере метод `$_isa` аналогичен стандартному `isa`: он проверяет, является ли `$foo` объектом класса `Class`, с тем лишь отличием, что если $foo не является объектом, то вместо исключения возвращается `undef`.

- App::cpanmw

Утилита `cpanmw` является обёрткой для `cpanm` и делает симпатичную подсветку вывода, если терминал поддерживает цвета. Подсветка работает в том числе и на платформе *Windows*.

- Net::DNS::Native

`Net::DNS::Native` неблокирующийся ДНС-резолвер, который использует вызов `getaddrinfo` вашей системной библиотеки. Разрешение имени происходит в выделенной нити, позволяя основной нити программы выполнять свою работу.

- Mesos

`Mesos` — это обвязка к разделяемой библиотеке *Mesos* проекта Apache для управления кластером серверов. Модуль позволяет создавать распределённые приложения (фреймворки), запускающие задачи на динамическом пуле серверов, управляемом *Apache Mesos* с поддержкой изоляции и выделения ресурсов.

- Plugin::Loader

Модуль `Plugin::Loader` позволяет просто и безопасно решать задачу по поиску и загрузке модулей-плагинов. Данная задача, наверно, один из самых распространённых случаев использования строкового `eval` неискушёнными разработчиками.

Обновлённые модули

- Mango 1.14

Новый релиз неблокирующегося драйвера MongoDB `Mango` по всей видимости стал последним. Себастьян Ридель объявил, что в связи с тем, что разработка актуального драйвера требует высоких трудозатрат, а каждый релиз MongoDB ломает обратную совместимость, разработка `Mango` прекращена. Почти сразу появился форк проекта, но пока неясно, станет ли он преемником.

- HTTP::Tiny 0.50

В новом релизе лёгкого веб-клиента `HTTP::Tiny` исправлена работа `keep_alive` при создании нового процесса/нити в программе.

- DBIx::Class 0.082800

Новый релиз `DBIx::Class` со множеством исправлений ошибок.

- MaxMind::DB::Reader 1.000000

Вышел первый мажорный релиз `MaxMind::DB::Reader` для работы с базами данных геолокации IP-адресов MaxMind. Параллельно выпущена и XS-версия модуля, работающая в 100 раз быстрее.

- Template::Tooolkit 2.26

Новый релиз популярного шаблонизатора `Template::Tooolkit` теперь поддерживает *контурные* теги (*outline tags* %%), позволяя избавиться от нагромождения скобок:

```
%% IF some.list.size
  <ul>
%%   FOREACH item IN some.list
    <li>[% item.html %]</li>
%%   END
  </ul>
%% END
```

- Data::Dumper 2.154

Обновление для модуля `Data::Dumper` содержит исправление ошибки безопасности CVE-2014-4330, при разборе глубоко вложенной структуры происходили расходование всего пространства стека и крах приложения. Теперь модуль содержит переменную `Maxrecurse`, которая по умолчанию ограничивает максимальный уровень рекурсии значением в 1000.

■ *Владимир Леттиев*

7. Интервью с Леоном Тиммермансом

Леон Тиммерманс (Leon Timmermans) — Perl-программист, биолог по образованию, автор и сопровождающий многих популярных модулей на CPAN.

Когда и как научился программировать?

Вообще-то первым языком, который я по-настоящему выучил, был Javascript, причем еще тогда, когда никто его толком не использовал ни для чего серьезного (мне кажется, в районе 1999 года). По иронии судьбы, его сейчас используют все, а я не прикасался к нему в течение многих лет. После этого я познакомился с C, Java и Perl, хотя я уже не помню, в каком это было порядке (это была непрерывная последовательность).

Какой редактор используешь?

Vim. Я начал им пользовался когда начал использовать Linux и никогда не пробовал использовать что либо другой (я даже использовал его на Windows).

Когда и как познакомился с Perl?

Думаю, что где-то в 2000-2001 году я наткнулся на Perl-книгу и начал играться с языком на своей Linux-машине (там был установлен 5.005004). Только в 2008 году я начал загружать свои модули на CPAN и через несколько месяцев позже посетил свой первый Perl-воркшоп. Так я и попал в сети Perl-сообщества.

С какими другими языками интересно работать?

Мне очень нравится C++, особенно когда вышел C++11 (и этим летом C++14). Он во многом похож на perl, у него такая репутация устаревшего языка, но если присмотреться, он живой и в активной разработке. И точно также как Perl, этот язык относится к тебе как к взрослому: он не говорит, как что-нибудь сделать, но предоставляет богатый инструментарий для достижения своей цели. Не поймите меня неправильно, C++ капризный, и каждый опытный C++ про-

граммист испытывает разного рода раздражение по отношению к нему (большее, чем к другим языкам), но как по мне, он вполне хорош.

Моим другим основным языком является C. Сегодня в большинстве случаев я использую его для работы над ядром perl и XS-модулями. Мне он все также нравится как и раньше, но C++ позволяет мне решать проблемы более эффективным способом.

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

Его выразительность. Он дает мне возможность написать вещи несколькими словами, которые бы заняли целые абзацы на других языках. Его желание дать мне всю возможную мощь вместо нескольких опций, как мне что-то сделать.

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

В длительной перспективе — расширяемость. Возможность реализации вне ядра вещей, которые авторы не предусмотрели.

В короткой перспективе — хорошая поддержка распараллеливания. Мы живем в многоядерном мире, но только некоторые программы пользуются этим, потому как большинство языков программирования не позволяют этого сделать. Есть незаполненная ниша.

Как пришла в голову идея написать прагму experimental?

Я заметил, что включение экспериментального функционала выглядит несколько странным. И я думаю, что включение их по умолчанию с выводом предупреждения было бы более правильным подходом, чем добавлении двух строчек, чтобы сказать perl о том, что ты знаешь, что делаешь (это не perl-подход). Поэтому мне пришла в голову идея написать модуль, который бы говорил, что ты делаешь что-то рискованное, без попытки помешать тебе в этом.

Как объединяешь биологию и программирование?

У меня вообще биологическое образование, а не программисткое.

Во время моего исследования я решил смешать мое хобби и работу.

Грубо говоря, биоинформатика может быть разделена на вычислительную сферу (как, например, 3D-моделирование белка) и на сферу больших данных (как, например, анализ генома), хотя на практике обычно проблема относится к обеим сферам. Я работал над второй. Многие инновации за последние два десятилетия ускорили процесс накопления данных быстрее, чем увеличиваются возможности компьютеров и устройств хранения данных, это все рождает многие интересные проблемы.

Что не так с File::Slurp?

Есть три проблемы.

Первая — это интерфейс. В современной работе с файловой системой кодировка файла важна настолько же, насколько и его имя. Это было инновацией perl 5.8, но File::Slurp был написан до этой версии. Откровенно говоря, вся парадигма сильно устарела. `read_file( $filename, binmode => ":encoding(utf-8)")` слишком длинно, что приводит к тому, что кодировку все игнорируют. В File::Slurper это выглядит как `read_text($filename)`, что, по-моему, выглядит несколько лучше.

Во-вторых, реализация содержит ошибки. Так как модуль был написан до IO layers и пытается их переизобрести, и не очень успешно. В частности, он некорректно декодирует/кодирует не-utf-8 символы (как, например, UTF-16 и KOI-*). Также у него есть проблемы с правильной обработкой перевода строк. Все это из-за небольших оптимизаций для частных случаев (чтение бинарного файла в переменную), что сильно усложняет код. Я планирую добавить похожую, но правильную оптимизацию в ядро perl версии 5.22, чтобы закончить этот спор.

Третья проблема — это сопровождение. Дистрибутив не обновлялся в течение трех лет несмотря на ошибки, найденные более полутора лет назад. Он и до этого редко обновлялся. Я понимаю, что у всех программ есть ошибки, но я не хотел бы зависеть от авторов, которые настолько не заботятся об их исправлении.

Правильно ли использовать потоки в Perl?

Обычно нет.

Они не обязательно зло, но на практике никто толком не знает, как эффективно их использовать. Их модель ни для чего по-настоящему не годится, они не масштабируются на больших данных.

Я работал над модулем акторов, что лучше вписывается во внутренности perl, но доведение модуля до возможности использования другими программистами довольно сложно. Существующая на сегодняшний день непонятная ситуация со smart-matching, возможно, вынудит меня вообще пересмотреть весь интерфейс.

libperl++ это законченный проект? Каков его статус?

Законченный? К сожалению, нет. Должен признать, что это несколько заброшенный проект, хотя у меня есть планы по его воскрешению.

libperl++ был очень амбициозным проектом, который научил меня C++ и Perl API. По факту, это самый сложный проект, который я когда-либо писал. Настолько сложный, что я загнал себя в угол. Объединение шаблонов, множественного наследования и неявных преобразований сделало эту смесь взрывоопасной.

Возможно, мне стоит сделать версию 2.0 с более компактным и не таким магическим кодом.

Улучшилась ли Perl-документация с 2010 года?

Местами. Не настолько, насколько мне бы хотелось. Некоторые вещи были переписаны, как например, perlopentut, многие вещи остались неизменными. На это стоит направить поток мотивированных волонтеров.

Стоит ли советовать молодым программистам учить Perl сейчас?

Конечно. Но опять же, я призываю программистов учить множество языков, возможно, даже очень разных. Perl объединяет прак-

тичность и изобретательность, его хочется изучать и применять.

Вопросы от наших читателей

Будешь ли снова начинать писать блог?

Возможно. Пока не уверен. Мне обычно довольно сложно заканчивать свои статьи, несмотря на большое количество идей. Вообще, у кодирования больший приоритет, поэтому статьи остаются незаконченными.

■ *Вячеслав Тихановский*