Pragmatic Perl 9

pragmaticperl.com

Выпуск 9. Ноябрь 2013

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

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

Авторы статей: Наталья Анисимова, Владимир Леттиев, Сергей Можайский

Корректор: Андрей Шитов

Выпускающий редактор: Вячеслав Тихановский (vti)

Ревизия: 2014-11-29 16:14

Оглавление

1 От редактора

Приглашаем читателей к участию в Perl Golf — оригинальном конкурсе на самую короткую Perl-программу. Читайте условия нового конкурса в этом номере!

Предложить свою тему или же взять тему для написания статьи теперь можно через репозиторий журнала на GitHub.

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

Приятного чтения.

■ *Вячеслав Тихановский*

2 Использование HTML::FormFu при работе с Catalyst

Введение в использование `HTML::FormFu` под `Catalyst`. Очень простые примеры, комментарии. Руководство для начинающих. Как создать `Catalyst`-приложение с нуля.

Все примеры были выполнены специально для публикации, в Windows-среде, под Strawberry Perl. В среде UNIX развернуть все необходимое и заставить работать должно быть даже проще. Приведенные примеры кода будут работать и в том, и в другом случае — они достаточно простые, чтобы не зависеть от тонкостей использования среды.

Для того, чтобы получить более полное представление о работе `HTML::`

FormFu в Catalyst, создадим Catalyst-приложение с нуля. Более того, установим Catalyst под Windows. Можно было пойти более простым путем, и использовать уже готовое рабочее UNIX-окружение, но тогда оставалась вероятность, что в описании забудется какой-нибудь важный модуль, или аспект, без внимания к которому подключить HTML::FormFu будет не просто.

Введение

Что такое Catalyst?

Catalyst — это фреймворк для создания веб-приложений. Поддерживает концепцию MVC (Model-View-Controller).

`Catalyst` поставляется вместе со своим собственным HTTP-сервером, который можно использовать для разработки и тестирования. В боевых условиях, его используют в связке nginx + FastCGI, или Apache + mod_perl. Можно использовать другие серверы, но такие решения встречаются значительно реже.

Несмотря на регулярную критику, `Catalyst` остается самым мощным и популярным фреймворком в Perl-среде.

Что такое HTML::FormFu?

`HTML::FormFu` — это менеджер форм для Perl-приложений. Гибкий и мощный инструмент. Поддерживает все этапы работы с формами: создание форм, их валидацию, вывод ошибок, сохранение данных в БД. `HTML::FormFu` считается одной из самых

мощных и функциональных систем для работы с формами в Perl.

Как установить Catalyst под Windows

Запускаем `cpan` в Perl-консоли. Потом вводим первую команду:

```
force install Catalyst::Runtime
```

На этом этапе может случится первый fail — если пытаться установить `Catalyst` (ну правильно, документацию читают только слабаки) вот так:

```
force install Catalyst
```

`cpan` что-то долго думает, что-то закачивает, устанавливает, но в итоге все

заканчивается ошибкой. Устанавливать надо `Catalyst::Runtime`, а не `Catalyst`!

`force install` нужен, чтобы запретить `cpan` слишком много думать и обращать внимание на результаты тестов. При установке Perl-модулей под Windows — это необходимо.

Потом устанавливаем еще немного дополнительных модулей, без которых сложно построить минимальное `Catalyst`-приложение или не будет работать `HTML::FormFu`.

```
1 force install Catalyst::
     Controller::HTML::FormFu
2 force install Catalyst::Devel
3 force install DBD::mysql
4 force install DBI
5 force install DBIx::Class
```

```
6 force install Catalyst::Model::
      DBIC::Schema
7 force install Catalyst::View::TT
8 force install DBIx::Class::Schema
      ::Loader
```

Для того, чтобы беспроблемно установить `DBIx::Class::Schema::Loader` — желательно прописать в переменных окружения путь к вашему локальному mysql-демону (во время тестирования использовалась локальная mysql-БД).

```
1 force install MooseX::
      MarkAsMethods
2 force install Catalyst::Plugin::
      Unicode::Encoding
```

Как создать каркас Catalyst-приложения под Windows

Создаем директорию, в которой будет расположен проект. Например: `C:\Documents and Settings\username\www`.

Совет: создавая директорию, следует избегать русскоязычных имен в пути к вашему проекту. В дальнейшем это может привести к возникновению проблем.

Переходим в созданную директорию. В командной строке Strawberry Perl вводим:

```
1 catalyst.pl MyApp
```

В результате, в каталоге `www` будет создана директория с именем `MyApp`, содержащая каркас `Catalyst`-приложения.

Как создать контроллер

Создаем контроллер `Admin.pm`, в дальнейшем он нам пригодится. Для этого в Perl-консоли выполняем команду:

```
perl script/myapp_create.pl
    controller Admin
```

Остальные контроллеры создаем вручную.

Как создать View

В отличие от контроллера, view вручную лучше не создавать. В Perl-консоли выполняем команду:

```
perl script/myapp_create.pl view
    TT TT
```

Команда создаст файл TT.pm в директории /lib/MyApp/View/TT.pm. Модуль TT.pm требуется отредактировать, добавив настройки:

```
package MyApp::View::TT;
use Moose;
use namespace::autoclean;

extends 'Catalyst::View::TT';

__PACKAGE__->config(
    TEMPLATE_EXTENSION => '.tt',
    render_die => 1,
    CATALYST_VAR => 'c',
    ENCODING     => 'utf-8',

);

1;
```

Далее, открываем файл MyApp.pm и добавляем еще немного конфигурационных дан-

ных для TT в блоке `__PACKAGE__->config()`:

```
__PACKAGE__->config(
    # ...

    'View::TT' => {
        INCLUDE_PATH => [
            __PACKAGE__->path_to(
                'root', 'src' ),
        ],
    },
);
```

Кроме того, в блок `use Catalyst qw/.../;` добавляем `Unicode::Encoding`. Это делается для того, чтобы `Catalyst` смог нормально обрабатывать русскоязычные символы в шаблонах.

```
use Catalyst qw/
    # ...
    Unicode::Encoding
```

```
4 /;
```

Если `Unicode::Encoding` не подключить, в дальнейшем можно будет увидеть в логах ошибку:

```
1 [error] Caught exception in
     engine "Wide character in
     syswrite at C:/strawberry/
     perl/lib/IO/Handle.pm line
     474."
2 Terminating on signal SIGINT(2)
```

Настройки для HTML::FormFu

В файле `MyApp.pm` в блок `__PACKAGE__->config()` добавляем настройки:

```
1 __PACKAGE__->config(
2     # ...
3
```

```
    'Controller::HTML::FormFu' =>
        {
        'model_stash' => {
            schema => 'DB'
        },
        constructor => {
            tt_args => {
                ENCODING => 'UTF
                    -8',
            }
        }
    }
);
```

Как создать модель

В Perl-консоли выполняем команду:

```
perl script/myapp_create.pl model
    DB DBIC::Schema MyApp::Schema
   ::DB create=static "dbi:mysql:
```

```
    test" "root" ""
```

После этого в блоке `connect_info` файла `lib/MyApp/Model/DB.pm` добавляем параметр `mysql_enable_utf8`:

```
1 connect_info => {
2     dsn => 'dbi:mysql:test',
3     user => 'root',
4     password => '',
5     mysql_enable_utf8 => 1
6 }
```

Этот параметр позволит корректно отображать данные из таблиц БД. Разумеется, если данные хранятся в кодировке UTF-8.

Как запустить сервер Catalyst под Windows

В консоли выполняем команду:

```
perl script/myapp_server.pl
```

Это позволит запустить специальный web-сервер. Для боевого использования он не подходит, а для тестирования новых возможностей — вполне. В эту же консоль будет печататься вывод отладочной информации сервера.

В браузере вводим адрес:

```
http://localhost:3000
```

Создаем интерфейс администратора с помощью HTML::FormFu и Catalyst

Панель администратора — это скрытый от обычных посетителей блок сайта, который позволяет владельцу осуществлять управление контентом сайта и принципами его отображения. Например, настроить список используемых виджетов, добавить новые публикации, разместить баннеры и т.п.

Почти вся админка — это бесконечное число разнообразных форм. Использование менеджера форм тут более чем оправдано.

Поэтому в качестве простого примера создадим примитивный интерфейс администратора:

- главную страницу панели администратора;
- страницу со списком публикаций на сайте;
- страницу для редактирования статей (на основе `HTML::FormFu`);
- страницу для создания новой публикации (на основе `HTML::FormFu`).

Кроме того, позволим пользователю удалять статьи.

Дамп БД

Для начала нам потребуется база данных. Ниже — дамп БД, которая использовалась для приведенных примеров. Установить БД, если ее у вас нет, лучше еще до начала установки `Catalyst` с его модулями для

создания моделей.

Теперь просто создаем таблицу и следим за кодировками, дабы избежать проблем в будущем. Как современные люди, мы выбираем UTF-8.

```
CREATE DATABASE IF NOT EXISTS `test`;
USE `test`;

CREATE TABLE IF NOT EXISTS `articles` (
  `id` int(10) NOT NULL AUTO_INCREMENT,
  `name` varchar(255) DEFAULT NULL,
  `full` text,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=3 DEFAULT CHARSET=utf8;

INSERT INTO `articles` (`id`, `
```

```
    name`, `full`) VALUES
12   (1, 'name', 'text'),
13   (2, 'name2', 'текст');
```

Главная страница панели администратора

Для максимально удобной работы с `HTML::FormFu` в `Catalyst`, существует модуль `Catalyst::Controller::HTML::FormFu`. Если планируем в создаваемом контроллере использовать менеджер форм, подключаем `Catalyst::Controller::HTML::FormFu` с помощью `extends`.

Создать форму можно либо определив ее параметры в непосредственно в коде, либо — по конфигурационному файлу. Конфигурационный файл предпочтительней.

Во-первых, это все-таки элементы HTML-кода, и уже давно стало хорошим правилом отделять разметку от кода. Во-вторых, при создании панели администратора в боевых условиях форм будет очень много, и удобнее хранить их отдельно.

Чтобы создать форму, нужно задать в атрибутах `Catalyst action` — `:FormConfig`. Если не указать путь к форме в атрибуте `:FormConfig`, система решит, что в качестве имени формы следует использовать имя `action`. Поиск файла будет осуществляться примерно по такому пути: `root/forms/controller_name/action_name.yml`.

Наличие атрибута `:FormConfig` говорит системе, что требуется создать объект формы и разместить его в хранилище `Catalyst` — `$c->stash->{form}`. Чтобы вывести форму на html-странице, доста-

точно добавить в шаблон инструкцию: `[% form %]`. `form` — это полностью готовый блок HTML-кода формы.

Проверить, была ли отправлена форма и корректны ли ее данные можно с помощью метода `$form->submitted_and_valid`.

Другой метод позволит получить данные из полей формы: `$form->param_value('field_name')`. Кроме того, можно получить значения полей формы с помощью `$c->req->param('field_name')`.

```
1 package MyApp::Controller::Admin;
2 use Moose;
3 use namespace::autoclean;
4
5 BEGIN { extends 'Catalyst::
      Controller'; }
6
```

```
7  sub index :Path :Args(0) {
8      my ( $self, $c ) = @_;
9
10         $c->stash->{template} = '
               admin/index.tt';
11         $c->forward('View::TT');
12 }
13
14 __PACKAGE__->meta->make_immutable
       ;
15
16 1;
```

Шаблон `/root/src/admin/index.tt`

Переходим в директорию `root`. Создаем в ней каталог `src`. В каталоге `src` создаем каталог `admin`.

Не забываем, что все шаблоны должны

быть в кодировке UTF-8 .

```
<h1>Панель администратора</h1>
<ul>
<li><a href="[% c.uri_for( '
    articles' ).path_query %]">
    Список публикаций</a></li>
</ul>
```

Для наглядности TT-шаблоны не содержат никакого HTML-кода, кроме самого необходимого.

Работа с публикациями

Модуль /lib/MyApp/Controller/Admin/Articles.pm

Один модуль будет отвечать и за отображение списка публикаций, и за создание новых, редактирование старых, и за удале-

ние статей.

```
1  package MyApp::Controller::Admin
       ::Articles;
2  use Moose;
3  use namespace::autoclean;
4
5  BEGIN { extends qw/
6      Catalyst::Controller::HTML::
           FormFu
7  / }
8
9  =head2 index
10
11 Отображение страницы со списком
       статей
12
13 =cut
14
15 sub index :Chained('') :Path :
       Args(0) {
16     my ( $self, $c ) = @_;
17
```

```
    $c->stash->{articles_list} =
       [$c->model('DB::Article')
       ->search(
        {}
    )->all];

    $c->stash->{template} = '
       admin/articles.tt';
    $c->forward('View::TT');
}

=head2 remove

Удаление статьи по ее
   идентификатору в БД

=cut

sub remove :Chained('') :Path('
   remove') :Args(1) {
    my ( $self, $c, $id ) = @_;

```

```
35      $c->model('DB::Article')->
           find({ id => $id})->delete
           ;
36      $c->res->redirect($c->uri_for
           ('/admin/articles'));
37  }
38
39  =head2 update
40
41  Данный блок кода отвечает за
       отображение страницы для
       редактирования статьи,
42  и за обновление статьи в БД.
43
44  =cut
45
46  sub update :Chained('') :Path('
       update') :Args(1) :FormConfig(
       'article/update.yml') {
47      my ( $self, $c, $id ) = @_;
48
49      my $form = $c->stash->{form};
50
```

```
51      if ($form->
            submitted_and_valid) {
52          eval {
53              my $obj = $c->model('
                    DB::Article')->
                    update_or_create({
54                   id => $form->
                        param_value('
                        id'),
55                   name => $form->
                        param_value('
                        name') ||
                        undef,
56                   full => $form->
                        param_value('
                        full') ||
                        undef
57              });
58          };
59
60          $c->warn('DB error') if
                $@;
61
```

```
62      } else {
63          my $article = $c->model('
               DB::Article')->search(
                {
64              id => $id,
65          } )->first;
66
67          $c->stash->{form}->
               default_values({
68              'name' => $article->
                   name,
69              'full' => $article->
                   full,
70              'id' => $article->id
71          });
72      }
73
74      $c->stash->{template} = '
           admin/article/update.tt';
75      $c->forward('View::TT');
76  }
77
78  =head2 create
```

```
79
80 Страница для создания новой
       статьи. Если пользователь
       нажал кнопку "Сохранить",
81 статья отправляется в БД, затем
       клиента перенаправляют на
       страницу со списком статей.
82
83 =cut
84
85 sub create :Chained('') :Path('
       create') :Args(0) :FormConfig(
       'article/create.yml') {
86     my ( $self, $c, $id ) = @_;
87
88     my $form = $c->stash->{form};
89
90     if ($form->
           submitted_and_valid) {
91         eval {
92             my $obj = $c->model('
                   DB::Article')->
                   create({
```

```
93                  name => $form->
                        param_value('
                        name') ||
                        undef,
94                  full => $form->
                        param_value('
                        full') ||
                        undef
95              });
96          };
97
98          $c->warn('DB error') if
                $@;
99
100         $c->res->redirect($c->
                uri_for('/admin/
                articles'));
101     } else {
102         $c->stash->{template} = '
                admin/article/create.
                tt';
103         $c->forward('View::TT');
104     }
```

```
105 }
106
107 __PACKAGE__->meta->make_immutable
    ;
108
109 1;
```

`default_values` — позволяет задать значения по умолчанию полям формы, то, что увидит пользователь, если откроет страницу с формой.

Можно использовать метод `$form->submitted`, чтобы понять, была форма отправлена или нет. Форма может быть отправлена, но не пройти валидацию. Метод `$form->submitted_and_valid` учитывает и отправку, и благополучное прохождение валидации.

```
<html xmlns="http://www.w3.org
   /1999/xhtml">
<head>
    <meta http-equiv="Content-
       Type" content="text/html;
       charset=utf-8" />
    <title>Формочка</title>
    <link rel="stylesheet" type="
       text/css" href="/static/
       style.css" />
</head>
<body>

<table cellspacing="0" id="
   result_list">
[% FOREACH article =
   articles_list %]
    <tr>
    <td>
        <a href="[% c.uri_for( '
           update', article.id ).
           path_query %]">[%
           article.name %]</a>
```

```
14          <a href="[% c.uri_for( '
               remove', article.id ).
               path_query %]">Удалить
               </a>
15      </td>
16      </tr>
17  [% END %]
18  </table>
19  <a href="[% c.uri_for( 'create' )
       .path_query %]">Добавить новую
       статью</a>
20
21  </body>
22  </html>
```

```
1  <html xmlns="http://www.w3.org
       /1999/xhtml">
2  <head>
3      <meta http-equiv="Content-
          Type" content="text/html;
```

```
        charset=utf-8" />
    <title>Формочка</title>
    <link rel="stylesheet" type="
        text/css" href="/static/
        style.css" />
</head>
<body>

Update article

[% form %]

</body>
</html>
```

```
<html xmlns="http://www.w3.org
    /1999/xhtml">
<head>
    <meta http-equiv="Content-
        Type" content="text/html;
```

```
        charset=utf-8" />
    <title>Формочка</title>
    <link rel="stylesheet" type="
        text/css" href="/static/
        style.css" />
</head>
<body>

Create new article

[% form %]

</body>
</html>
```

CSS-стили для формы `/root/static/style`

Добавим совсем простую CSS-таблицу, которая позволяет аккуратно вывести элементы формы. В дальнейшем, CSS можно

усложнить, создавая с его помощью профессиональное оформление всей панели администратора.

```
form {
  width: 40em;
}

.submit {
  display: block;
}

label {
  display: block;
}
```

YAML-конфиг для формы `/root/forms/art`

По умолчанию `Catalyst`-приложение будет искать формы в директории `root/forms`. Для работы с конфигурационными файлами `Catalyst::Controller::HTML`

::FormFu использует Config::Any. Соответственно, хранить конфигурационную информацию о формах кроме YAML-формата можно в XML, JSON, конфигах в стиле Apache, Perl-коде и т.п.

Переходим в директорию root. Создаем в ней каталог forms. В каталоге forms создаем каталог article и файл update.yml.

```
1  ---
2  attributes:
3      id: element_form
4
5  auto_fieldset:
6      attributes:
7          class: module aligned
8
9  ---
10 elements:
11     - type: Hidden
12       name: id
13
```

```
    - type: Text
      name: name
      label: Название статьи

    - type: Textarea
      name: full
      label: Текст статьи

    - type: Submit
      name: submit
      value: Сохранить
```

```
---
attributes:
    id: element-form

auto_fieldset:
    attributes:
        class: module aligned

---
elements:
```

```
    - type: Text
      name: name
      label: Название статьи

    - type: Textarea
      name: full
      label: Текст статьи

    - type: Submit
      name: submit
      value: Сохранить
```

Вот и все. Простой прототип панели администратора с использованием `HTML::FormFu` готов. Теперь на основе полученных результатов можно пробовать усложнять формы, вводить в работу сложные поля и правила валидации, добавлять JavaScript для работы со сложными элементами, CSS для создания современного интерфейса.

В данном руководстве ничего не сказано про работу `HTML::FormFu` с БД своими средствами. Честно говоря, автора эти возможности не впечатлили. Возможно, примеры, которые встречались в сети, были слишком простыми. Но пока не видно никакой разницы, использовать для сохранения в БД средства `HTML::FormFu` или стандартный запрос с помощью `DBIx`.

■ *Наталья Анисимова*

3 Обзор изменений Perl 5.19.5

20 октября 2013 г. был выпущен очередной релиз Perl для разработчиков 5.19.5. Особенным этот релиз делают несколько интересных возможностей, которые были добавлены в язык и будут включены в последующую стабильную версию, став ключевыми изменениями нового Perl.

Экспериментальная постфиксная запись разыменования

Данная экспериментальная возможность позволяет использовать постфиксную запись для разыменования ссылки с помощью комбинации символов сигила и звездочки, например:

```
@{ $ref->{foo} }
```

теперь можно записать так:

```
$ref->{foo}->@*
```

В обоих случаях будет возвращён массив.

Постфиксная запись более удобна тем, что позволяет читать и записывать выражение слева направо, без необходимости возвращаться к началу выражения, чтобы обособить его символами `@{}`.

Аналогично массиву определено разыменование и других типов:

```
$sref->$*;  # тоже самое, что и ${ $sref }
$aref->@*;  # тоже самое, что и @{ $aref }
$href->%*;  # тоже самое, что и %{ $href }
```

```
4 $cref->&*;  # тоже самое, что и
      &{ $cref }
5 $gref->**;  # тоже самое, что и
      *{ $gref }
```

Поскольку данный функционал является экспериментальным, то для того, чтобы его можно было использовать, требуется явно включить с помощью `use feature` , а также подавить предупреждения об использовании экспериментальной возможности:

```
1 no warnings "experimental::
      postderef";
2 use feature "postderef";
```

В случае, если требуется, чтобы постфиксная запись также интерпретировалась и внутри строк, необходимо дополнительно подключать возможность `postderef_qq`:

```
no warnings "experimental::postderef";
use feature "postderef", "postderef_qq";

print "$ref->{foo}->@*";
```

Постфиксную запись можно использовать и для получения срезов массивов и хешей:

```
$aref->@[1,2];          # тоже самое, что и @{$aref}[1,2]
$href->@{'foo','bar'};  # тоже самое, что и @{$href}{'foo','bar'}
```

Кроме того, в постфиксной записи доступен и новый синтаксис среза, появившийся в Perl 5.19.4, для получения хеш-среза:

```
%h = $href->%{'foo','bar'} # тоже самое, что и %h = %{$href}{'foo', 'bar'};
```

```
%h = $aref->%[3,4,6];       # тоже самое, что и %h = %{$aref}[3,4,6];
```

Рассмотрим пример обхода сложной структуры:

```
for my $key (keys %{$href}) {
    for my $subkey (keys %{$href->{$key}}) {
        push @{$href->{$key}->{$subkey}->{foo}},
             @{$href->{$key}->{$subkey}->{bar}}{'a','b'}
    }
}
```

Количество фигурных скобок просто зашкаливает, снижая читаемость и повышая шанс на ошибку. В новом синтаксисе всё выглядит уже не так плохо:

```
for my $key ( keys $href->%* ) {
    for my $subkey ( keys $href
        ->{$key}->%* ) {
         push $href->{$key}->{
            $subkey}->{foo}->@*,
             $href->{$key}->{
                $subkey}->{bar}->@
                {'a','b'}
    }
}
```

В целом постфиксная запись разыменования должна работать во всех случаях, в которых работает и обычное разыменование в блоке (*ци́ркумфикс*), и является эквивалентной. Однако в некоторых ситуациях отличия могут появиться, например:

```
no strict 'refs';

my $foo = 'bar';
${my $foo = 'baz'} = 'qux';
print $foo; # напечатает 'bar'
```

```
6
7 my $foo = 'bar';
8 (my $foo = 'baz')—>$* = 'qux';
9 print $foo; # напечатает 'baz';
```

В первом случае переменная `$foo` в блоке изолированна и не влияет на внешнюю переменную, а в постфиксной записи `$foo` переопределяется.

Альтернативная запись прототипа функции

В рамках работы над внедрением сигнатуры функции в ядро Perl, которую ведёт *Peter Martini*, в новом релизе Perl появилась возможность задавать прототип функции через специальный атрибут функции — `prototype`. Таким образом, следующие

объявления прототипов функции эквивалентны:

```
sub foo($$){
    ...
}

sub foo : prototype($$){
    ...
}
```

Объявление прототипа поддерживается и на уровне модуля `attributes`, т.е. запись:

```
use attributes __PACKAGE__, \&foo
    , "prototype(\$\$)";
```

также задаст прототип функции `foo`.

Никаких существенных преимуществ новая запись не даёт, но открывает возможности для дальнейшей работы над расширением синтаксиса функций.

Unicode 6.3

В Perl 5.19.5 включена поддержка новой версии Unicode 6.3 и, поскольку новая версия стандарта 7.0 ожидается только через год, то данная версия скорее всего и войдет в Perl 5.20. Основным новшеством в версии 6.3 стало появление специальных управляющих символов, позволяющих задавать изолированные по направлению фразы текста, т.е. появляется возможность вставлять символы, направление письма которых может отличаться от направления окружающего текста.

Установка

Как обычно, протестировать новую версию Perl можно с помощью `perlbrew`:

```
perlbrew install 5.19.5
perlbrew switch 5.19.5
```

Устанавливайте, тестируйте и делитесь впечатлениями!

■ *Владимир Леттиев*

4 Обзор CPAN за октябрь 2013 г.

Рубрика с обзором интересных новинок CPAN за прошедший месяц.

Статистика

- Новых дистрибутивов — 236
- Новых выпусков — 939

Новые модули

- ReturnValue Модуль `ReturnValue`, позволяющий возвращать структурированное значение об ошибке или успехе из функций. Может быть удобен, чтобы унифицировать подход к

реализации функций, которым требуется возвращать сложный набор информации, включая информация об успехе, возвращаемых данных и прочей мета-информации о результатах. Вероятно это попытка сделать замену устаревшему и глючному `Return::Value`.

- Devel::Confess Модуль `Devel::Confess` — это небольшое подспорье при отладке, заставляет `die` и `warn` делать более детальный выхлоп, включающий трассировку вызовов.

```
$ perl -d:Confess -e 'sub foo { die }; foo'
Died at -e line 1.
    main::foo() called at -e line 1
```

- Linux::Socket::Accept4 Модуль является обёрткой к системному вызову

`accept4(2)`, доступный в системах GNU/Linux.

- constant::override Оказалось, что константы обладают существенным недостатком — их значения нельзя изменить. Поэтому появился модуль, который позволяет переопределять или удалять константы.

- DateTime::Moonpig Этот модуль является обёрткой к `DateTime`, но с более вменяемым интерфейсом. Так, методы, которые меняли объект при вызове (`add_duration`, `set_*`), теперь вызывают фатальную ошибку (защита от выстрела себе в ногу). Также переопределены операции сложения и вычитания, позволяя упростить вычисления длительности между датами и расчёта новых дат. Добавлено несколько новых методов.

- Crypt::DSA::GMP Новая реализация стандарта DSA для проверки цифровой подписи. Модуль реализован на чистом Perl и является обратно совместимым с модулем `Crypt::DSA`. В отличие от `Crypt::DSA`, данный модуль при возможности следует стандарту *FIPS 186-4* и поддерживает несколько новых методов хеширования.

- Redis::Fast Очередная реализация клиента для redis, созданная на базе C-клиента `hiredis`, совместимая с модулем `Redis`. Утверждается, что в бенчмарках данный модуль на 50-300% быстрее `Redis`.

- Router::Boom Matsuno Tokuhiro выпустил альтернативу своему собственному модулю `Router::Simple`. Алгоритм поиска маршрута дан-

ного модуля имеет сложность *O(1)*, позволяя в бенчмарках на порядок обгонять `Router::Simple`.

- mop Вышел пробный релиз модуля реализации *Metaobject Protocol* (MOP) для Perl 5, который имеет все шансы для попадания в базовый дистрибутив Perl.

- CBOR::XS Perl-модуль с реализацией компактного бинарного формата данных CBOR для межсистемного обмена. Упор в стандарте сделан на простоту кодировщика/декодировщика, компактность сообщений и однозначное (де)кодирование различных типов данных без схем описаний.

Обновлённые модули

- AnyEvent::Filesys::Notify 0.24 В новом релизе модуля для мониторинга изменений директорий и файлов произошёл переход с `Moose` на `Moo`.
- XML::TreeBuilder 5.0 Вышел новый мажорный релиз модуля `XML::TreeBuilder`, в котором была добавлена поддержка XML-каталогов и раскрытия XML-сущностей (entities). Кроме того, были удалены излишние зависимости (для xt-тестов).
- Path::IsDev 1.000000 Первый стабильный релиз модуля для определения, является ли заданный путь каталогом, в котором находятся разрабатываемый исходный код. Изменения значительны, и вполне вероятно, что ранее использовавший этот модуль

код может оказаться сломанным.

- Text::Xslate 3.0.0 Новый мажорный выпуск шаблонизатора Text::Xslate включает несколько исправлений ошибок, в том числе проблему с использованием ключей хешей в многобайтовой кодировке в шаблонах.

- HTML::Mason 1.52 Вышло обновление для стабильной ветки 1.x модуля `HTML::Mason` с исправлением ошибок. В частности исправлена проблема, связанная с рандомизацией ключей хеша в Perl 5.18+.

- Mouse 1.13 Исправлена сборка модуля на Perl 5.19.4.

- Test::Mocha 0.21 Несмотря на название, это довольно интересный модуль для быстрого создания заглушек

и мок-объектов при тестировании, похожий на *Mockito* из Java-мира. В новом релизе модуль полностью избавился от зависимости от `Moose`.

- Promises 0.04 В новом релизе модуля реализации Promises для Perl функция `when` объявлена устаревшей, и взамен предлагается использовать `collect`. Связано это в первую очередь с конфликтом с существующим ключевым словом `when` в Perl 5.10+.

- SNMP 5.0404 Непонятно, почему этот модуль до сих пор не удалили со CPAN, ведь актуальная версия 5.7.2 поставляется только в составе продукта Net-SNMP. Модуль на CPAN был обновлён с версии 5.0401 до 5.0404; судя по изменениям в коде был приложен патч для устранения старой уязвимости с переполнением буфера

CVE-2008-2292.

- Pinto 0.091 В новой версии модуля Pinto для управления репозиториями добавлена поддержка внешних https-репозиториев.
- JSON::XS 3.01 Вышел новый мажорный релиз модуля JSON::XS. В новой версии появилась возможность (де)сериализовать Perl-объекты за счёт использования нестандартного расширения к JSON-синтаксису. Окончательно выкинуты функции `from_json` и `to_json` (используйте `encode_json` и `decode_json`). Модуль получил зависимость от `Types::Serialiser`, который предоставляет булевы константы `true` и `false`.

■ *Владимир Леттиев*

5 Интервью с Marc Lehmann. Часть 1

Marc Lehmann — автор AnyEvent, Coro, common::sense, JSON::XS и многих других популярных модулей на CPAN.

Когда и как начал программировать?

Я начал программировать, когда мне было около девяти лет с какой-то восьмибитной платой, где нужно было вводить инструкции в шестнадцатиричном коде. Мои родители где-то «услышали», что компьютеры «плохо влияют на развитие» и поэтому не хотели покупать мне «настоящий» компьютер, но та плата их не смущала. К счастью, через несколько лет они купили мне C-128, а вскоре после этого Amiga 500, который был идеальным компьютером для знакомства с устройством

ОС, низкоуровневым программированием и многим другим. Первые компьютеры позволяли мне понять, как устроено железо и как программировать на низком уровне, но Amiga OS была настолько же сложной, как и ядро современной операционной системы. Когда Amiga стала уже слишком старой и медленной (несмотря на апргейд платы до 68030), я перешел на упрощенную платформу PC+DOS, а в 1993 на HP/UX и затем, почти сразу, на GNU/Linux.

Если говорить про языки, я начал с машинного кода, затем переключился на BASIC и 6502, Modula-2 и m68k, затем на Turbo Pascal + x86 и в конечном итоге на Perl + C на HP/UX и GNU/Linux.

Конечно, я никогда не прекращал учиться программировать, например, в этом году я

выделил время, чтобы по-настоящему выучить Javascript (по сравнению с тем, чем я занимался) и до сих пор постоянно учу новые подходы в программировании на Perl и C, несмотря на то, что на этих двух языках я программирую достаточно давно. Perl особенно хорош в своей способности удивлять меня.

Если говорить о том, как я научился программировать, то я читал документацию, которая шла либо с языком, либо с компьютером, который у меня появлялся, и пытался разобраться. Многому научился на информатике, но скорее не программированию, поэтому считаю, что я выучил многие вещи методом самостоятельных проб и ошибок.

Самым важным способом обучения для меня было (и до сих пор остается) чтение

чужого кода. Мне кажется, что я в двадцать раз больше читаю кода, чем пишу. Или даже соотношение может быть несколько большим.

На сегодняшний день я часто учу новые языки, просто читая их спецификацию, чтобы избежать бредовой информации, разбросанной по интернету. Читаю случайные блоги с базовой информацией и затем спецификацию, чтобы все подкорректировать. Это отлично работает.

Какой редактор используешь?

Исключительно VIM. Я честно пытался выучить Emacs, даже купил книгу “Learning GNU/Emacs”, чего я обычно не делаю, чтобы что-то выучить. До сего дня я четко помню, на какой странице мой мозг бросил исключение и перезагрузился. Если

выразиться иначе, то я, возможно, понимаю, что Emacs более крутой редактор, но мой мозг на базовом уровне несовместим с ним, и я остаюсь с VI до конца моих дней.

Поэтому мне кажется, что все холиворы вокруг редакторов совсем не о том: дело в твоем мозге, он либо VI-образный, либо Emacs-образный. Не важно каким бы редактор хорошим не был, возможно вам он вообще не подходит.

Между тем, до VIM я пользовался некоторое время joe — для того, кто пришел из Turbo Pascal/Turbo C, joe удобен тем, что повторяет те же горячие клавиши. Я вспомнил joe, потому что у него до сих пор есть одна исключительная особенность — возможность работать с файлами, которые не помещаются в памяти. Поэтому когда нужно интерактивно отредактировать

файл размером 60 ГБ — joe прекрасно с этим справляется.

Когда и как познакомился с Perl?

Это было в 1993 на HP/UX — я искал удобный язык для Unix. Я был потрясен интернетом и свободно доступной информацией, «стандартами» (RFC) и тем фактом, что практически все (конфигурационные файлы, протоколы, такие как FTP) в Unix и в интернете представлено в виде текста.

Perl 4 вышел с хорошей документацией (и бесплатной :), и так я приобщился к нему. Когда перешел на Perl 5 в 1995, я просто перечитал все man-страницы по порядку, сильно впечатлился и до сих пор не могу отойти.

Моей первой Perl-программой был ftp-

клиент на curses, который мог загружать файлы в фоновом режиме — очень полезная фишка, когда средняя скорость была 200 байт/с. После того как клиент стал популярным, это позволило мне проводить различные исследования в области безопасности (получать доступ к аккаунтам пользователей, разумеется только с целью обучения). Когда в университете узнали об этом, они предложили мне работу, что, кажется, сильно мне пригодилось.

С какими другими языками нравится работать?

«Нравится» – довольно растяжимое понятие :)

Я часто объединяю Perl и C++ (если возможно), или C (если нет). Мне очень нравится XS за его мощь и относительную простоту

(то, что получается на выходе, не процесс изучения), поэтому я бы описал мой основной язык как «Perl+XS».

И так как я играюсь со многими языками (для развлечения или по работе), я не могу сказать, что они мне действительно нравятся. Я периодически копаюсь с posix shell, разными диалектами ассемблера и javascript и стараюсь не смотреть на php/python/ruby... код ни под каким предлогом, с переменным успехом правда.

Так что по-настоящему мне нравится только Perl.

Что, по-твоему, является наиболее сильным преимуществом Perl?

Я не уверен, что у Perl до сих пор есть явное преимущество, но среди языков этой кате-

гории я предпочитаю Perl за его ядро и гибкость интерпретатора.

Возьмем, к примеру, Coro. Меня поражает тот факт, что можно добавить такой функционал поверх оригинального интерпретатора как простое расширение языка. Попытки делать нечто похожее в других языках, например, в Python, обычно выливаются в переписывание или разработку нового интерпретатора.

Также есть множество «микро-особенностей», которые нельзя счесть за что-то большое, но они действительно таковыми являются. Например, `__END__` — не выглядит как что-то выдающееся, однако позволяет элегантно запустить perl-процесс, скажем, через ssh-соединение. `AnyEvent::Fork::Remote` использует эту особенность, но я пользовался этим и раньше в коммерче-

ских проектах. В других языках, например Python, нет эквивалента `__END__`, поэтому реализация `AnyEvent::Fork::Remote` потребует грязных хаков, включая временные файлы и прочее. В Perl решение простое и элегантное, и Perl полон таких время от времени полезных суперштук.

CPAN, например, крут, и в течение многих лет был определенно важным преимуществом, но на сегодняшний день этим не удивишь пользователей других языков, и качество многих модулей сильно хромает.

Но CPAN, ядро и гибкость, объединенные в один язык Perl, являются все еще уникальной комбинацией.

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

Не уверен, что у меня есть хороший ответ. Надеюсь, что языки будущего позволят мне делать, что я хочу, при этом не мешая.

Если говорить о ближайшем будущем, я надеюсь, что языки позволят просто обмениваться данными между разными процессами или разными нодами. Конечно, я работаю над этим в Perl (мой список дел довольно длинный — `libev`/`EV` были в моем списке в течение десяти лет, когда они стали достаточно приоритетными), но мне еще далеко до завершения.

Если говорить о далеком будущем, подозреваю, что языки будущего будут более графическими, управляемыми сознанием (и, возможно, контролирующими сознание), и настолько раздражающими, что я все еще буду писать на древнем Perl, когда у всех будут новые и блестящие

квантовые компьютеры, встроенные в их головы, которые слушают подсознание и сами пишут программы. Если, конечно, это будет экономически выгодно.

Но, если серьезно, было бы действительно хорошо иметь возможность просто объяснить задачу компьютеру (про себя, используя некие абстрактные мысли), и машина сама реализует решение и отладит его...

Ты написал несколько модулей для событийного программирования, которые стали очень популярными. Почему ты решил открыть их, и почему, по-твоему, они стали настолько распространенными?

Это сложный вопрос — проще ответить, почему бы так не произошло: я не гоняюсь за публичностью и славой.

Мне кажется, что публикация своего кода — это естественно. Альтруизм тоже играет важную роль — когда я думаю, что что-то может быть полезным (как например, «хотелось бы, чтобы это было уже кем-то написано») и у меня есть возможность, я просто обязан опубликовать код для общего блага. В конце концов, я видел, как другие люди делают то же самое — если кто-то публикует свой код, другие люди воодушевляются этим, и всем это идет на пользу.

Я стараюсь документировать свои модули большей частью для того, чтобы они были полезными кому-нибудь еще. Эта дополнительная работа позволяет сделать модуль полезным на практике, а не в теории. Документация — это единственная часть моего софта, которую я пишу для других. Код я пишу, потому что сам его использую.

Почему некоторые мои модули так распространены, я могу лишь догадываться, и обычно для меня это не так важно. Я даже не знаю, какие из них популярны, а какие — нет.

Если взять `AnyEvent`, у меня был партнер по разработке, который считал этот модуль настолько полезным, что он обязан был быть известным. И поэтому мы пытались написать несколько «необходимых» дополнительных модулей, как например `AnyEvent::HTTP`. Тяжело сказать, насколько это помогло, я не сильно старался, даже дополнительные модули были написаны, потому что были нужны лично мне.

Возможно, эти модули стали популярными, потому что я изучал событийное программирование в течение двух десятков лет, про себя возмущаясь различными

проблемами и отсутствием необходимой поддержки, и в результате, попытался реализовать все, чего мне не доставало и я считал базовым, без слишком сложного интерфейса.

По факту, в AnyEvent и EV можно увидеть процесс эволюции. У AnyEvent интерфейс похож на Event (методы). Позже EV научило меня, что достаточно минимального интерфейса (функции с несколькими фиксированными параметрами), и поэтому я переработал интерфейс AnyEvent в AE.

В конце концов, я надеюсь, что мои популярные модули стали таковыми, потому что помогли кому-то решить проблему, также как они помогли решить мою проблему. Хочу также надеяться, что это из-за моего стремления к качеству, которое делает эти модули популярными, но я всегда

думаю об этом в большом секрете :)

Можно ли коротко описать почему Coro это единственные настоящие треды в Perl?

По-видимому, коротко — нет.

Определяющим свойством тредов в других языках (Python или C) является общее адресное пространство (код и данные). Когда ничего общего нет, это называется «процессами».

«Модель конкурирующих тредов» в Perl, ithreads, использует подобные треды в C для эмуляции Unix-процессов в Perl. На уровне Perl вы получаете (полную ошибок) модель процессов, и разделение переменных или кода между ithread настолько же (не-)эффективно и неестественно, как и

разделение их между настоящими процессами, с тем лишь исключением, что настоящие процессы не должны эмулировать MMU (Memory Management Unit — блок управления памятью, — *прим. перев.*). Разделение объектов и кода даже не реализовано (объекты, основанные на массивах и хешах, приходят из других тредов пустыми). Разделение уже существующих структур данных вообще невозможно, и так далее.

Поэтому, выражение «единственные настоящие треды в Perl» не было задумано для спора, но как очевидная констатация факта. Однако в течение прошлых лет люди снова и снова давали разные определения тредам и затем оспаривали это выражение.

По существу, все эти споры были вариациями на тему «тред это что-то, что работает

параллельно с другими тредами, потому Coro не настоящие треды, а ithreads — да», не учитывая, что: а) придется не признавать тредами pthreads, треды в Python, треды в Ruby и большинство других реализаций, так как почти все они не работают параллельно; и б) процессы также подпадают под это определение тредов, что делает само определение не очень полезным (уже есть термин для процесса — «процесс»).

В то время как процессы действительно являются тредами выполнения, общее значение, которое люди вкладывают в треды в императивных языках программирования, означает «множество тредов выполнения разделяют общее адресное пространство», и поэтому Coro единственные настоящие треды в Perl, разделяющие естественным образом код и данные.

Почему был написан common::sense?

Я всегда хотел, чтобы предупреждения были полезными, но многие предупреждения в Perl только мешают (мне?) или совершенно некорректны в мелких деталях, делая все предупреждения бессмысленными.

Отсутствие `common::sense` означало бы либо дублирование настроек во всех моих программах и модулях, или отключение вообще всех предупреждений.

Поэтому причиной написания `common::sense` было упрощения поддержки — выделить общий код из разных модулей и поместить его в отдельную общую библиотеку, которую можно везде использовать. Если посмотреть на документацию к `common::sense`, можно увидеть, что код не очень простой и несколько раз менялся.

Без `common::sense` мне бы пришлось каждый раз выпускать изменения в своих модулях, что не сильно бы помогало пользователям.

Что там произошло между `JSON::XS` и сортировкой хешей в Perl?

Есть безусловно несколько правильных точек зрения на эту тему, но моя заключается в следующем.

Много лет назад в `CGI.pm` была ошибка, которая могла привести к исчерпанию ресурсов системы, и по какой-то причине разработчики Perl 5 подумали, что внесение изменений в ядро языка было лучше, чем исправление ошибки в модуле, или введение системы лимитов для таких случаев.

В то время мне это показалось странным, в

конце концов, другие языки с подобными типами данных, не рандомизировали их по причинам безопасности, и поэтому C++ и другие языки страдают от тех же самых «уязвимостей» (или по крайней мере исправляют проблему исчерпания ресурсов, а не симптомы).

Тем не менее, в документации было сказано, что порядок ключей будет тем же самым в пределах одного запуска программы, и это было таковым для более чем сотни модулей (в то время на CPAN было мало связанных с этим проблем, так как только изредка программы зависели от порядка хеша между разными запусками, так как не было подобной гарантии).

И только недавно это рандомизация была признана недостаточной (или сломанной), и была написана другая система, когда тот

же самый хеш мог возвращать ключи в разном порядке даже в пределах запуска одной программы, что в общем-то не сильно требовалось для решения первоначальной проблемы с безопасностью.

Не было никакого цикла изменений — документация и код были исправлены, и (к счастью) было создано много патчей для важных модулей (не JSON::XS, а, например, LWP), которые бы не работали со следующим выпуском perl.

Мое участие во всем этом выражалось в сомнении насчет необходимости столь резкого изменения. Оно было бы действительно необходимым в случае серьезной уязвимости, но представленные аргументы для меня не были достаточными (несколько других языков решают это «старым» и надежным способом, и не было ни одного приме-

ра эксплуатирования этой уязвимости). И до сих пор никто, у кого бы я не спросил, не ответил мне, почему не была решена изначальная проблема (или хотя бы проверена).

Единственный ответ, который я получил (от релиз-менеджера perl) был следующим: «Это изменение решает проблему, и на данный момент у нас нет лучшего патча». Это вполне разумно, но только звучит как постфактум.

Это не первая «особенность», которая ломает предыдущий код без хорошей на то причины в последних релизах perl, и, может это только я, но для меня исправление ошибок и стабильность гораздо важнее новых клевых штук.

По факту Perl пришел к более частым выпускам — примерно один мажорный релиз

в год. К сожалению, обратная совместимость перестала быть приоритетом, поэтому на сегодняшний день мне приходится мириться с регулярными поломками, что возникают в моих модулях и на которые я должен тратить много своего времени. Отчеты об ошибках несовместимости часто служат целью для высмеивания и преувеличения, и это вынуждает меня жестко критиковать весь процесс разработки.

Если проследить за различиями между стабильностью и обратной совместимостью от perl 5.000 до текущих дней, повторяющиеся поломки в текущем Perl довольно разительны.

Можешь привести несколько примеров, когда `App::Staticperl` может быть полезным?

Я часто отправляю клиентам скомпилированный с помощью staticperl исполняемый файл вместе с другими Perl-файлами — часто они не знают Perl и не смогут справиться с необходимостью установки большого количества дополнительных модулей. С работающим staticperl создание бинарного файла ненамного сложнее обычного запуска программы, и даже если человек знает, как устанавливать модули, очень удобно иметь возможность сразу запустить программу.

На системах со статической линковкой (не glibc и не windows) можно даже создать исполняемый файл, который будет работать везде без зависимостей — например, я часто рассылаю GNU/Linux-бинарники, который запускаются на x86- и amd64-системах с новым ядром, вне зависимости от libc или других вещей (alpine linux — хорошая обо-

лочка для создания таких файлов).

Иногда клиенты ничего не хотят знать про Perl, и staticperl позволяет мне спрятать perl в исполняемый файл или динамическую библиотеку, чтобы не пугать пользователей. Удивительно насколько быстрым и удобным может быть Perl, когда пользователи не знают, что программа написана на нем, а не на C :)

Также удобно встраивать perl в программу без многих лишних файлов. staticperl для меня создает .h и .c файлы, а затем компилирует и линкует их с libperl. У меня получается полнофункциональный интерпретатор Perl, плюс библиотеки Perl и свои модули, без каких-либо внешних файлов и без каких-либо зависимостей в файловой системе. Все в одной C-программе.

Это актуально на практике еще и тогда, когда я могу встроить perl в тех случаях, когда бы встраивал lua.

Также часто возникает необходимость протестировать необычные опции сборки Perl разных версий с помощью staticperl. Возможно, это я такой: staticperl не был задуман для этого. Часто я слышу много восторженных отзывов о perlbrew, поэтому сперва попробуйте его, если вам нужно собрать несколько perl-версий.

К сожалению, все особенности staticperl достаются огромной ценой. Несколько кривых модулей (в основном `Module::Build`) делают использование staticperl не таким простым, а доступным только экспертам, которые понимают, как компилируется Perl, в чем разница между статической линковкой и статическим исполняемым

файлом. У большинства людей нет желания или необходимости знать это. Я иногда думаю, что я единственный, который регулярно встраивает Perl (для меня это естественно, но очевидно, что не для всех).

Если вы все же продвинутый пользователь, создание независимых исполняемых файлов со staticperl становится развлечением. Это гораздо быстрее и легче (для меня), чем использование, например, `PAR::Packer`. Также у staticperl-файлов больше шансов фактически заработать, судя по моему (возможно, однобокому) мнению. Если бы `PAR::Packer` работал надежно, не было бы причины писать `App::Staticperl` или `Urlader`.

(В качестве рекламы, `Urlader + Perl ::LibExtractor` это еще один способ

запаковать perl, чем я и пользуюсь на Windows. Это все также быстрее и проще, чем `PAR::Packer`, и работает с любой программой, не только Perl. Да и, для меня, концептуально проще).

Ты довольно сильно оптимизируешь свои программы. Это обычно необходимость или просто привычка?

Вообще-то я думал, что большинство моих программ не сильно оптимизированы, но я понимаю, почему может сложиться такое впечатление.

Во-первых, у меня достаточно большой опыт не только в программировании, но также в том, что быстро, а что — нет, и поэтому мне действительно сложно писать не эффективно если, особо не напрягаясь, я могу достичь лучшего результата. Поэтому

многие оптимизации, которые вы видите, из-за привычки или потому, что я хотел избежать плохого — по моим меркам — кода.

Во-вторых, я очень сильно оптимизирую базовые и часто используемые функции (библиотеки, модули и тому подобное), поэтому при написании фактической программы, использующей эти библиотеки, я пишу более понятный, компактный и, возможно, не совсем быстрый код.

Посмотрите на это под другим углом: можно написать приложение на C, можно написать на Perl. При прочих равных на C оно будет быстрее, но довольно сложным в написании. Поэтому, возможно, стоит написать его на Perl — что будет удобнее, и так как Perl уже достаточно оптимизирован на уровне C, приложение будет достаточно

быстрым.

Поэтому основным принципом в работе должен быть «используй наиболее удобный язык, который достаточно быстрый», и поэтому сильно оптимизированные C-библиотеки в Perl делают его выбор наиболее частым из-за удобства.

Если соединить «используй подходящий для задачи язык» и Perl+C, то этим обычно покрываются все задачи, потому что Perl хорош в тех вещах, где C отстает, и наоборот, что в результате выливается в лучшее из обоих языков.

В реальности я обычно слишком ленив для оптимизации, и поэтому оптимизирую, когда это действительно необходимо. Это позволяет мне не напрягаться и писать может и не быстрый, но простой и понятный вы-

сокоуровневый код.

Однако, у скорости всегда меньший приоритет, чем у корректности.

О том где работает Марк, каковы его мысли насчет будущего Perl, конференций и сообщества, почему он до сих пор использует CVS и другие интересные ответы читайте в следующем номере.

■ *Вячеслав Тихановский*

6 Как я решал Perl Golf от PragmaticPerl

В прошлом номере журнала был объявлен конкурс Perl Golf, в котором я решил поучаствовать.

Условия задачи здесь приводить не буду, они есть по ссылке http://pragmaticperl.com/issu 08-perl-golf.html

Сначала я набросал прототип решения задачи. Алгоритм был выбран довольно простой — вначале собираем входные данные в одну строку, потом регулярным выражением проходим по горизонтали строку и заменяем на пробелы нужные цифры. Для обработки вертикалей входные данные разбирались на вертикальные строчки, по ним проходило то же регулярное выражение, что и для горизонталей, а затем строки поворачи-

вались обратно.

Вот что получилось в первой итерации:

```
1  #!perl
2  sub h {
3      ($x) = @_;
4      while ($x =~ /(\d)(\s*)(?=\d)
          /g)
5      {    # проходим по
          горизонтали
6          $f = $1;    # первое
              число
7          $s = $2;    # промежуток
8          $' =~ /^(.)/
9            ; # разбираем POSTMATCH
                , получаем второе
                число
10         if ($f + $1 == 10 || $f
              == $1) {
11             $x =~
12               s/$f$s$1/ $s / #
                    заменяем числа
```

```
                    пробелами
13          }
14      }
15      $x;
16  }
17
18  sub v {
19      ($y) = @_;
20      @e = ();
21      for $i (0 .. 8) {
22          $l = '';
23
24          # получаем вертикальные
              строки
25          map { $l .= substr($y, $i
              + $_ * 9, 1) } 0 ..
              $w;
26
27          # проходим по ним
              процедурой h, как для
              горизонталей,
28          # и записываем в массив
              @e
```

```
29          push @e, h($l);
30      }
31      $y = '';
32
33      # получаем строку в формате
            исходных данных
34      for $i (0 .. $w) {
35          map { $y .= substr($_, $i
                , 1) } @e;
36      }
37      $y;
38  }
39  ($c = join '', <>) =~ s/\n//g; #
        объединяем в одну строку,
        убираем пробелы
40  $w = length($c) / 9 - 1;        #
        получаем высоту поля
41  while (1) {
42      $d = v(h($c));              #
            проходим по горизонтали и
            вертикали
43      last if $d eq $c;           #
            если ничего не изменилось
```

```
        — завершаем
44      $c = $d;
45  }
46  for ($d =~ /[ \d]{9}/g)
47  {    # разбиваем на строки по 9
        символов
48      # и выводим их на экран,
            если строка содержит хоть
             одну цифру
49      print "$_\n" if /\d/;
50  }
```

Примечание: код здесь и далее отформатирован для улучшения читаемости — в реальном гольфе это все записано в одну строку.

372 символа! Это очень много по меркам гольфа. К счастью, тут есть что улучшать.

- Параметр процедуры получаем как

`$x=pop;` вместо `($x)=@_;` — экономия в один символ.

- Используем постфиксные if и for везде где только возможно — это дает экономию в 3 символа на каждое условие или цикл.
- Если процедура определена выше, то скобки при вызове не обязательны — `h($l)` можно записать как `h$l`.
- `length($c)` можно записать как `$c=~y///c`, а это на один символ короче.
- Условие выхода пишем прямо в `while` — `last` не нужен.
- Использование `$_` позволяет заметно сократить программу. К примеру, `$x=~s/a/b/` заменяется на `s/a/b/` — экономия в 4 символа.
- Вместо `$'=~/^(.)/` и получения результата в `$1` пишем `$'=~/./` — результат будет в специальной переменной `$&`. Еще 3 символа.

- Поскольку нам все равно нужен диапазон `0..$w` (`$w` — это высота поля), его лучше получить заранее и сохранить в массив — получится сэкономить порядка 7 символов.
- Флаги командной строки (http://perldoc.p заметно упрощают жизнь. Добавим `-0` (в итоге входные данные сразу приходят как одна строка), `-l` (при выводе в конце строк автоматически добавится `\n`) и `-n` (не нужно отдельно обрабатывать STDIN, он сразу будет в `$_`).
- Для экономии можно использовать встроенные переменные Perl. К примеру, переменная `$,` (разделитель вывода для print) и другие ей подобные хороши тем, что после них можно не ставить пробел: `$,ne$_` вместо `$a ne$_`.

После всех этих оптимизаций получаем 299 символов:

```
#!perl -ln0
sub h {
    $_ = pop;
    while (/(\d)(\s*)(?=\d)/g) {
        $f = $1;
        $s = $2;
        $' =~ /./;
        s/$f$s$&/ $s / if $f + $&
            == 10 || $f == $&;
    }
    $_;
}

sub v {
    $y = pop;
    @e = ();
    for $i (0 .. 8) {
        $l = '';
        $l .= substr $y, $i + $_
            * 9, 1 for @r;
```

```
19              push @e, h $l;
20          }
21          $y = '';
22          for $i (@r) {
23              $y .= substr $_, $i, 1
                    for @e;
24          }
25          $y;
26      }
27      s/\n//g;
28      @r = 0 .. y///c / 9 - 1;
29      while ($, ne $_) {
30          $, = $_;
31          $_ = v h $,;
32      }
33      for (/.{9}/g) {
34          print if /\d/;
35      }
```

Оптимизируем дальше:

- Зачем нам процедура `v`, если мы все равно вызываем ее только один раз? Поместим ее код внутрь основного цикла `while`, это сэкономит немало символов.

- Тернарные операторы короче, чем `if` — используем их везде где возможно.

- Регулярное выражение в процедуре `h` можно существенно оптимизировать — в цикле достаточно пройтись по всем цифрам, а промежуток и вторую цифру получить из `postmatch`. В результате код:

```
while (/(\d)(\s*)(?=\d)/g) {
    $f = $1;
    $s = $2;
    $' =~ /./;
    s/$f$s$&/ $s / if $f + $
       & == 10 || $f == $&;
}
```

превращается в:

```
$f = $&, $' =~ /\d/, $f + $&
     == 10 || $f == $& ? s/$f
    (\s*)$&/ $1 / : 0
  while /\d/g;
```

- Получение высоты поля и удаление переносов строки можно улучшить — ведь высота поля равна количеству переносов строки, и наше

```
s/\n//g;
@r = 0 .. y///c / 9 - 1;
```

можно записать как:

```
@r = 0 .. (s/\n//g) - 1;
```

- Разбиение на вертикали тоже можно улучшить, если немного вспомнить математику:

```
$e["@-" % 9] .= $& while /./
    g;
```

С помощью регулярного выражения проходим все символы, сам символ (совпадение шаблона) хранится в переменной $&. Текущая позиция совпадения шаблона находится в массиве @-, а поскольку там всего один элемент, мы его преобразуем в скаляр, взяв в кавычки. Остаток от деления позиции совпадения на 9 даст нам номер колонки.

- Вычитать 1 из высоты поля совсем необязательно, алгоритм все равно будет игнорировать `undef`-значения.
- Несмотря на то, что мы не используем `strict`, мы можем использовать `my` для того, чтобы очистить массив.

После всех этих оптимизаций удалось достичь 214 символов.

Потом, изучив внимательно `perlrun`, я обнаружил ключик `-p`, который заставляет Perl выводить значение `$_` после завершения программы. Но для этого нужно было не удалять из него переводы строк, а также учитывать что длина каждой строки теперь 10 символов. Так удалось перейти очередной лимит и дойти до 199 символов:

```
#!perl -n0p
sub h {
    my $_ = pop;
    $f = $&, $' =~ /\d/, $f + $&
       == 10 || $f == $& ? s/$f(\
       D*)$&/ $1 / : 0
      while /\d/g;
    $_;
}
while ($, ne $_) {
    my @e;
    $e["@-" % 10] .= $& while /./
       gs;
    $, = '';
```

```
12      for $i (0 .. y/\n//) {
13          $, .= substr h($_), $i, 1
                for @e;
14      }
15      $_ = h $,;
16      s/ {9}\n//;
17  }
```

Похоже, это финиш и оптимизировать этот алгоритм дальше некуда. Но что, если попробовать другой алгоритм?

В принципе, большую часть программы занимает код для получения вертикалей. Вот если бы найти регулярное выражение, которое могло бы обрабатывать вертикальные совпадения, да еще и любой высоты… Я уже пробовал это сделать на начальном этапе, но вместо этого сосредоточился на улучшении алгоритма. Как оказалось, зря :)

Задача оказалась не такой уж простой, но после нескольких попыток у меня получилось подобрать подходящий регэксп. Код сразу же уменьшился до 172 символов, и похоже что это не предел:

```
#!perl -n0p
while ($, ne $_) {
    $, = $_;
    $f = $&, $p = $', $p =~ /\d/,
      $f + $& == 10 || $f == $& ?
          s/$f(\D*)$&/ $1 / : 0,
      $p =~ /((.{9}\s)*.{9})(.)/s
          , $s = $1,
      $f + $3 == 10 || $f == $3 ?
          s/$f$s$3/ $s / : 0
      while /\d/g;
    s/ {9}\n//;
}
```

В дальнейшем у меня вышло сделать одно регулярное выражение, под которое попадали и горизонтальные, и вертикаль-

ные совпадения. Но как оказалось, код можно улучшить еще больше, если снова вспомнить математику. В условии гольфа сказано — “если сумма равна 10 или обе цифры одинаковы”. То есть успешное совпадение для цифры X — это когда на второй позиции стоит либо X, либо 10-X. А это означает что нам не нужно проверять ни сумму, ни совпадение цифр — достаточно прогнать регулярное выражение для довольно ограниченного количества вариантов.

В результате получилось всего 96 символов:

```
#!perl -0p
while ($, ne $_) {
    $, = $_;
    for $a (1 .. 9) {
        for $b ($a, 10 - $a) {
            s/$a((.{9} )*.{9}|\D
                *)$b/ $1 /s;
```

```
7           }
8       }
9       s/ {9}\n//;
10  }
```

Финальная оптимизация:

- Два цикла здесь не нужны — поскольку совпадением является одна цифра, мы можем использовать `[$a$b]` как шаблон для второй цифры.
- Наконец, можно избавиться от промежуточной переменной, в которой хранится предыдущее состояние поля. Достаточно просто считать, сколько раз сработали регулярные выражения, и выполнять цикл, пока это число не станет равным нулю. В качестве начального значения я решил использовать переменную $/

— по умолчанию она равна \n и при проверке даст true.

И вот он, окончательный результат — 90 символов:

```
#!perl -0p
while ($/) {
    $/ = s/ {9}\n//;
    for $a (1 .. 9) {
        $b = 10 - $a;
        $/ += s/$a((.{9} )*.{9}|\
            D*)[$a$b]/ $1 /s;
    }
}
```

Вот такая история.

Во время решения гольфа я узнал немало нового про возможности Perl, о которых раньше даже не слышал или же никогда

не пользовался. Надеюсь, и вы тоже узнали из этой статьи что-то новое для себя :)

P.S. Спасибо Владимиру Леттиеву за организацию конкурса!

■ *Сергей Можайский*

7 Perl Golf

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

Цифры

То ли задание оказалось невообразимо сложным для читателей, то ли глупым и неинтересным (ни одной «звезды» не поставили репозиторию), то ли виной банальная лень, но было сделано лишь два(!) форка репозитория задачи на github golf-08. Pull-request'ы с решением были получены

вообще только от одного человека — *Сергея Можайского* technix. Сергей грамотно подошёл к выполнению задания и регулярно отправлял улучшенные варианты своего решения. Помимо выполнения задания Сергей улучшил и тест для проверки решения, и сделал важные замечания по критериям проверки.

Например, пришлось явно изменить методику подсчёта длины скрипта. Первоначально в условии задачи говорилось, что в длину скрипта не входит первая строка с шебангом. Но от этого правила пришлось отказаться и учитывать также длину всех параметров, передаваемых в шебанге, таким образом, из подсчёта длины исключается только путь к интерпретатору и начальные символы `#!`. Основная причина — это наличие классного хака в Perl, позволяющего в параметрах передать часть кода

скрипта. Сергей показал пример такого кода, я лишь приведу упрощённый для понимания пример:

```
#!/usr/bin/perl -i$x="yet\
    x20another\x20perl\x20hacker";
    print$x
eval$^I
```

Данный скрипт выведет фразу `yet another perl hacker`, при этом длина такого скрипта по методике, не учитывающей первую строку, всего 7 байт. Хитрость состоит в том, что у perl существует параметр `-i`, который позволяет редактировать файлы, имена которых переданы в командной строке, при этом значение после параметра задаёт расширение, которое будет добавляться для оригинальной копии редактируемого файла. Данная опция и поведение досталось perl в наследство от *sed*. Значение параметра можно получить

внутри программы в специальной переменной `$^I`. Таким образом, если часть кода вынести в параметр для `-i`, то его затем можно выполнить в программе через обычный `eval`.

Решение

Единственное предложенное решение закономерно оказалось победителем турнира. Длина решения составила всего 90 байт! Для удобства восприятия будет продемонстрирован отформатированный вариант

```
#!perl -p0
while ($/) {
    $/ = s/ {9}\n//;
    for $a ( 1 .. 9 ) {
        $b = 10 - $a;
```

```
6           $/ += s/$a((.{9} )*.{9}|\
                D*)[$a$b]/ $1 /s;
7       }
8   }
```

Опция `-p` задаёт режим, в котором программа оборачивается в такой цикл:

```
1 while (<>) {
2       ... # код скрипта
3 } continue {
4       print or die;
5 }
```

Т.е. поток `STDIN` читается и сохраняется в переменной `$_`, а после того как весь `STDIN` будет прочитан, происходит печать на `STDOUT` содержимого `$_`.

Опция `-0`, после которой нет параметра, означает, что символом-разделителем строк `$/` становится символ с кодом `0x00`.

Поскольку в нашем случае такого символа на входе точно нет, то это значит, что сразу будет прочитан весь `STDIN` в переменную `$_`.

Затем следует цикл `while`, который выполняется, пока значение `$/` является истинным. Интересно, что первоначальное значение `$/=chr(0)` является истинным, поэтому цикл выполняется.

В строке 3 происходит удаление пустых строк (т.е. строк, где все цифры заменены пробелами). Если удаление произошло, то в `$/` записывается `1`, в противном случае — пустая строка.

В строке 4 начинается цикл по обходу всех возможных вариантов цифр от 0 до 9. Для каждой такой цифры вычисляется парное число (в сумме дают число 10). И

вся суть алгоритма удаления пар заложена в регулярном выражении, которое ищет вхождения текущего числа, за которым следует ноль или несколько последовательностей из девяти произвольных символов и пробела (это проверка совпадения по вертикали), или за ним следует ноль или несколько нецифровых символов `\D*`, и затем идёт само число или его пара `[$a$b]`. Всё это заменяется строкой, в которой парные символы заменены пробелами. Модификатор `s` позволяет рассматривать в качестве любого символа `.` также символы переноса строки `\n`.

Если регулярное выражение выполнилось, то к переменной `$/` добавляется 1, в противном случае добавляется 0. Таким образом цикл продолжает свою работу, пока все возможные пары цифр и пустые строки не будут удалены из переменной `$_`. Когда цикл

завершится, переменная $_ будет выведена на экран.

Моё личное решение составило 126 байт. Используется схожий алгоритм, только я не догадался его так же здорово оптимизировать, как это сделал Сергей:

```
#!/usr/bin/perl -p
$d .= $_
}{
while ( $s ne $d ) {
    $s = $d;
    for ( 1 .. 9 ) {
        $j = 10 - $_;
        $d =~ s/$_(\s*|.{9}(\s
            .{9})*)($_|$j)/ $1 /sg
            ;
    }
}
$_ = $d =~ s/(^\s{9}\n|\n\s{9})//
    rg
```

Используется переменная `$d` для считывания всего `STDIN`, затем символ эскимо `}{`, чтобы разорвать внешний цикл `while`, который формирует опция `-p`. Во внутреннем цикле я проверяю, есть ли отличия в строке копии, чтобы выяснить была ли произведена замена. Ну и совсем избыточно выглядит регулярное выражение удаления пустых строк, которое к тому же использует модификатор `r`, который доступен только в Perl 5.14+.

Победитель

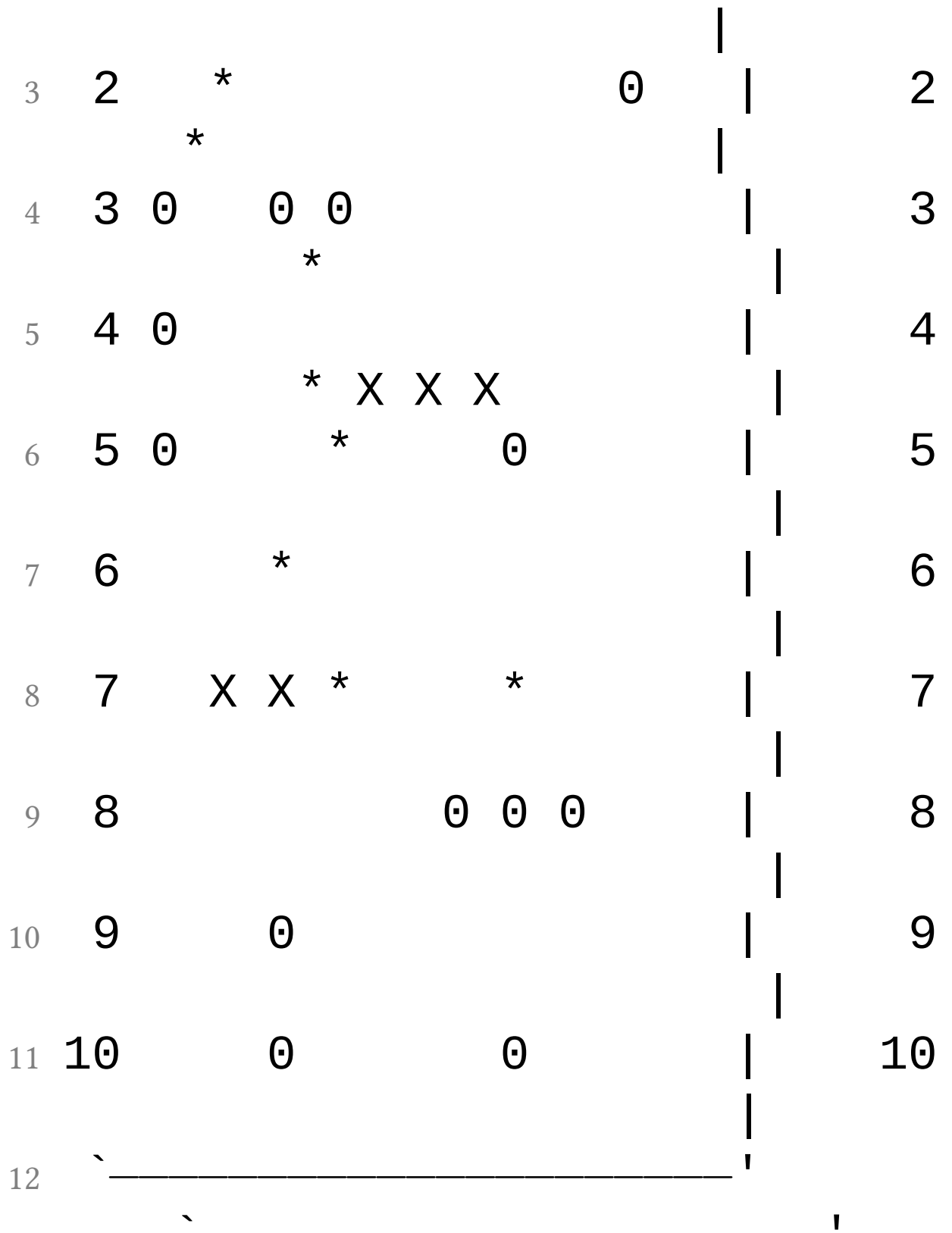
Поздравляем Сергея Можайского с победой в турнире по Perl Golf восьмого выпуска журнала «Pragmatic Perl»! Желаем дальнейших успехов и удачи на этом нелёгком поприще.

Новое задание — «Морской бой»

В этом выпуске я хочу предложить вам поиграть в «Морской бой». Всем с детства известны правила этой игры. На карте из 10x10 клеток размещаются десять кораблей: один четырёхпалубный, два трёхпалубных, три двухпалубных и четыре однопалубных корабля. Корабли могут располагаться как вертикально, так и горизонтально, но при этом не должны соприкасаться углами. Ну и дальше игроки вслепую обмениваются ударами, сообщая координаты удара и получая информацию о результате: мимо, ранен, убит.

В ASCII-графике это может выглядеть примерно так:

```
1  .—а—б—в—г—д—е—ж—з—и—к—.     .—а—б—в—г—д—е—ж—з—и—к—.
2  1   0 0 0 0             |     1 *
```


2
3
4
5
6
7
8
9
10
11
12

Попробуйте реализовать такой код, который получив на `STDIN` данные в виде ASCII-символьного поля 10x10, с изображёнными на нём результатом обстрела в виде * — промах, X — палуба поражённого корабля и пробел — неизвестная территория, попытается выполнить расчёт и поразить одну палубу ещё не обнаруженного четырёхпалубного корабля, выдав на `STDOUT` ту же самую карту, с проставленными символами @ в точках вероятного нахождения линкора. Гарантируется, что на входном поле нет недобитых кораблей.

Тестовый код будет проверять не только то, что вам удалось поразить линкор, но также просуммирует количество неуспешных попыток поражения (лишних @) в виде штрафных баллов, которые будут увеличивать длину вашего скрипта на 10 байт за каждый бал.

Репозиторий нового турнира доступен на github golf-09. Сделайте форк, создайте в директории `script` файл `your_github_login.pl` с вашим вариантом решения. Проверьте, что ваш вариант проходит тесты (с помощью команды `prove`) и сделайте pull request в основной репозиторий. Помните, что чем раньше вы опубликуете свой результат, тем больше шансов на победу.

Проверяться решения будут на последней стабильной версии Perl, т.е. 5.18.1. Приём решений закончится 30 ноября 2013 г. в 23:59:59.

Дерзайте, юнги! Победителя ждут слава и адмиральские погоны.

■ *Владимир Леттиев*