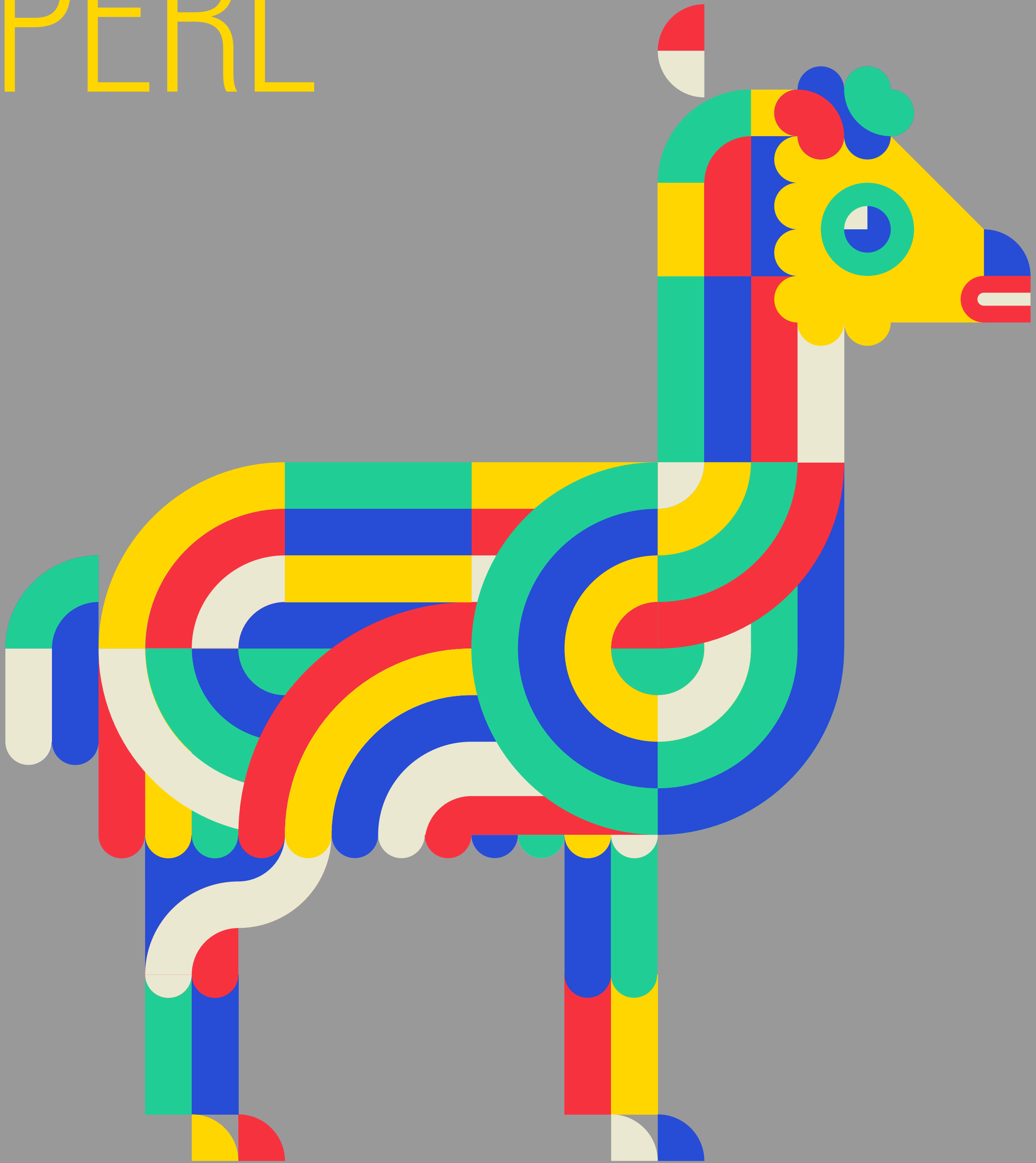
PRAGMATIC PERL

22



12/2014

pragmaticperl.com

Pragmatic Perl 22

pragmaticperl.com

Выпуск 22. Декабрь 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: Андрей Шитов, Вячеслав Коваль, Сергей Романов, Наталья Савенкова, Владимир Леттиев

Обложка: Марко Иваньк

Корректор: Андрей Шитов

Выпускающий редактор: Вячеслав Тихановский

Ревизия: 2014-12-22 14:51

© «Pragmatic Perl»

Оглавление

1 От редактора 1

2 Анонс воркшопа Saint Perl 2014 2

3 ООП. Основные паттерны проектирования. Реализация в Perl 3

4 Perl 6 XXI века 13

5 DBIx::Class. Сборник рецептов 46

6 Обзор CPAN за ноябрь 2014 г. 76

7 Интервью с Олафом Алдерсом (Olaf Alders) 82

1. От редактора

Новости журнала: мы преодолели рубеж в 1000 подписчиков! Также, как вы уже успели заметить, несколько расширился анонс номера, который мы отправляем по email и rss. Теперь, как нам кажется, гораздо удобнее сразу переходить к интересующей вас статье.

Продолжаем работать над сайтом журнала. У каждой статьи в оглавлении можно увидеть автора и ее краткое описание. В скором времени планируем добавить страницу автора, где можно увидеть все его/ее статьи.

Мы открыты к предложениям от читателей! Сообщите нам, чего вам не хватает или что можно сделать удобнее.

С начала декабря и до католического рождества некоторые популярные Perl-проекты запускают так называемый Advent Calender (Рождественский календарь), где каждый день публикуется новая статья. В этом году (на момент выпуска журнала) ведутся следующие календари: Perl, Perl6, Dancer, Возможно, что появятся и другие, следите за новостями.

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

Приятного чтения.

■ *Вячеслав Тихановский*

2. Анонс воркшопа Saint Perl 2014

В шестой раз подряд мы с радостью приглашаем всех любителей и профессионалов мира Perl в Санкт-Петербург на ежегодный воркшоп ***Saint Perl****!*

Основная часть конференции пройдёт 21 декабря на Василиевском Острове — в этот раз в роли гостеприимной принимающей стороны выступает компания T-Systems.

Но в этом году мы решили не ограничиваться только сессией докладов и блиц-докладов. В субботу, 20 декабря, состоится первый в Санкт-Петербурге Perl-хакатон! Свои идеи можно добавить на вики-странице на сайте конференции. О месте проведения хакатона будет дополнительно объявлено там же.

Специально для Pragmatic Perl спешим поделиться приятным инсайдом: предварительно своё участие в воркшопе подтвердил брайан ди фой — автор многочисленных публикаций по Perl и регулярный спикер на всевозможных конференциях и митапах. Интервью с ним в журнале: часть 1 и часть 2. Мы очень надеемся, что у брайана (именно так, с маленькой буквы!) всё сложится с визой, и вы сможете лично пообщаться с ним уже совсем скоро в Петербурге.

Сайт конференции: http://event.yapcrussia.org/saintperl6/. Мы с нетерпением ждём ваших докладов и идей для хакатона!

Электронная почта организаторов: saintperl_orgs@yapcrussia.org

До встречи в Питере!

■ *Сергей Романов*

3. ООП. Основные паттерны проектирования. Реализация в Perl

Материал статьи для уровня Beginners. Здесь не будет Moose, только чистый Perl. Предполагается, что какое-то ООП в Perl уже знакомо

Паттерны это стандартные приемы, решающие небольшую конкретную задачу. Это не инструкция, как писать код, а схема или принцип организации кода, модулей и т. п. Уверена, что если вы их не знаете на уровне диаграмм UML, то встречали в коде. Этот небольшой обзор познакомит с самыми простыми, полезными и часто используемыми паттернами.

Singleton (Одиночка)

Порождающий паттерн. Используется в случае, когда в системе должен быть только один экземпляр какого-то класса. Например, подключение к базе, распарсенный файл конфигурации и т. д. Но при этом вы не хотите таскать с собой какие-то глобальные переменные. Невероятно удобен для отложенных инициализаций тех же конфигов.

Реализация

Пусть у нас будет какой-то абстрактный класс с именем `MyClass`.

```
1 package MyClass;
2 use strict;
3 our $singleton = undef;
4
5 sub new {
6     my $class = shift;
7     return $singleton if defined $singleton;
8     my $self = {};
9     $singleton = bless($self, $class);
10    $singleton->init();
11    return $singleton;
12 }
```

```
13
14 # other methods
15 sub init {
16
17     #...
18 }
19
20 1;
```

`$singleton->init();` — вот тут, к примеру, проводится какая-то инициализация (либо она может быть отложена до вызова конкретных функций).

Пример использования

```
1 use MyClass;
2 use strict;
3
4 sub f {
5     print MyClass->new()->{name}, "\n";
6 }
7
8 sub f2 {
9     print MyClass->new()->{name}, "\n";
10 }
11 my $obj = MyClass->new();
12 $obj->{name} = 'Bob';    # это не ООП!
13 f();
14 f2();
15 $obj->{name} = 'Mike';    # и это тоже
16 f();
17 f2();
```

На выходе

```
1 Bob
2 Bob
3 Mike
4 Mike
```

В результате вызова функций `f()` и `f2()` мы получим один и тот же созданный объект, ссылка на который хранится у нас в `$MyClass::singleton`, с ней можно работать напрямую, но это моветон и делать так не надо (за исключением ситуаций, когда требуется высо-

кая производительность, а использование аксессоров создаёт ощутимые накладные расходы).

Таким образом, можно в любом месте кода создавать объект через конструктор и не волноваться, что он каждый раз будет создаваться заново.

На CPAN, кстати, есть `Class::Singleton`, `MooseX::Singleton`, `Apache::Singleton` и еще куча других.

Abstract Factory (Абстрактная фабрика)

Порождающий паттерн. Берет на себя ответственность за создание объекта нужного класса. Мы просто обращаемся к ее конструктору, а какой нам вернуть объект, фабрика решает сама. Создаваемые объекты, конечно, должны быть из одного семейства и иметь идентичный интерфейс. То есть, они должны быть взаимозаменяемыми.

В качестве примеров использования: в номере 21 в статье Тестирование в Perl. Практика паттерн использован для создания объекта-логгера в зависимости от способа вывода: либо stderr, либо file. В более бизнесовом мире встречаются разные способы доставки (там все одинаковое, но разные формочки, разные коэффициенты какие-нибудь), разные форматы прайсов от поставщиков (у кого-то Excel, у кого-то XML), разные способы отправки уведомлений (e-mail, SMS).

У меня будет пример очень абстрактный, но очень понятный. Допустим, у нас есть ферма с животными. Нам, с точки зрения логики, все равно, какое животное будет создано, мы только задаем в параметрах, сколько у него ног. (В реальности значение количества ног мы получаем из внешнего конфига, а не задаем в коде).

Пример использования

```
1 use AnimalFactory;
2 my $animal_one = AnimalFactory->new(legs => 2);
3 print ref $animal_one, "\n";
4 my $animal_two = AnimalFactory->new(legs => 4);
5 print ref $animal_two, "\n";
```

```
$animal_one->walk();
$animal_two->walk();
```

На выходе

```
Chicken
Cow
```

Реализация

```
package AnimalFactory;
use Chicken;
use Cow;

sub new {
    my $class = shift;
    my $opt   = {@_};
    return Cow->new()     if $opt->{legs} == 4;
    return Chicken->new() if $opt->{legs} == 2;
}
1;
```

Тут важно понимать, что обращаясь к конструктору AnimalFactory, мы получаем объект класса вовсе не AnimalFactory, а того, который она решит создать.

Если нам понадобится класс `Snake`, то мы просто добавим логику его создания в `AnimalFactory`, как-нибудь так:

```
return Snake->new() if $opt->{legs} == 0;
```

Если вдруг `Cow` нужно будет заменить на `Horse`, это нужно будет сделать только в одном месте — в `AnimalFactory`, не затрагивая других участков кода.

Абстрактную фабрику стоит использовать там, где класс объекта зависит от каких-нибудь внешних факторов: пользовательских настроек, версии браузера, ОС и т. п.

(В некоторых случаях не очень хорошо, что мы подгружаем все возможные классы сразу через `use`, это можно изменить: внести внутрь конструктора и подключать классы через `require` уже после анали-

за параметров и до создания конкретного объекта.)

Template Method (Шаблонный метод)

Паттерн поведения. Паттерн используется для определения основного алгоритма для всех подклассов. Берем алгоритм, делим его на много мелких этапов, пишем в базовом классе, а все подклассы реализуют различающиеся части.

Самый простой пример: импорт товаров от поставщика. Нужно распарсить файл, пройти по всем товарам от поставщика, если товар найден — обновить его, если не найден — создать, подсчитать конечную стоимость, записать операцию с товаром в журнал, проделать что-нибудь еще с чем-нибудь.

Использование

```
1 my $import = ImportFactory->new(type => 'Bekka');
2 $import->do;
```

(Здесь я использую фабрику для создания нужного мне объекта по имени поставщика, от которого загружается файл.)

Но можно обойтись и без фабрики, а сделать вот так (хотя гибкость это явно снижает, но она и не всегда такая нужна):

```
1 my $type = 'Bekka';
2 my $import = $type->new();
3 $import->do;
```

Реализация

Допустим, у меня тут два поставщика: `Bekka`

```
1 package Bekka;
2 use base 'Import';
3
4 sub parse {
5
6     # parse Excel
```

```
7 }
8
9 sub count_price {
10
11     # price * 2
12 }
13
14 1;
```

который присылает файлы в Excel, и у которого цену из файла нужно увеличивать в два раза.

И `Pukka`, у которого файлы в XML, а цену нужно делить пополам:

```
1 package Pukka;
2 use base 'Import';
3
4 sub parse {
5
6     # parse XML
7 }
8
9 sub count_price {
10
11     # price / 2
12 }
13
14 1;
```

Оба эти класса имеют родителя `Import`, который и описывает основной алгоритм загрузки файла (`sub do`). В нем определяются все используемые методы, но работающие по какому-то умолчанию. (У методов, конечно, еще есть какой-нибудь код, но здесь он не нужен, поэтому его не привожу.)

```
1 package Import;
2 ...
3 sub do {
4     my $self = shift;
5     $self->parse();
6     while ($self->next) {
7         if ($self->find) {
8             $self->update;
9         }
10        else {
11            $self->insert;
12        }
```

```
13         $self->count_price;
14         $self->log;
15     }
16     $self->finish;
17 }
18 sub next;
19 sub find;
20 sub update;
21 sub insert;
22 
23 sub count_price {
24     my $self = shift;
25 
26     # use original price
27 }
28 
29 1;
```

Получается: фабрика создает нам объект нужного класса, основываясь на имени поставщика. Базовый объект для него описывает весь процесс импорта товара от *любого* поставщика. Объект конкретного класса переопределяет те методы, которые ему не подходят, на свою реализацию — в нашем случае методы `count_price` и `parse`.

Метод `do` из класса `Import` и есть наш шаблонный метод — он описывает шаблон поведения. И вовсе необязательно, что он должен его реализовывать. В реальности сложно найти задачи такого плана, которые могут быть удовлетворены поведением по умолчанию.

Удобно использовать констукцию `can` для методов, которые не обязательно должны быть в базовом классе, но могут быть в подклассах: `$self->do_smth if $self->can('do_smth')`, тогда метод будет вызваться только в том случае, если он реально определен. Это избавит от кучи пустого кода, а также позволяет писать довольно удобно хуки, типа:

```
1 $self->before_update() if $self->can('before_update');
2 $self->update();
3 $self->after_update() if $self->can('after_update');
```

Strategy (Стратегия)

Паттерн поведения. Другое название — Политика. Используется для взаимозаменяемости алгоритмов или их фрагментов. Например, когда у нас есть разные способы расчета скидки на заказ. (Пример высосан из пальца, и для таких случаев делать подобные схемы — роскошь. Но он прост и понятен.)

Использование

```
use DiscountFactory;
use Order;
my $order = Order->new();
$order->{summa} = 200; # так делать — не ООП! Это только для примера
my $discounter = DiscountFactory->new(type => 'Visa');
print $order->get_summa(discounter => $discounter), "\n";
$discounter = DiscountFactory->new(type => 'yandex');
print $order->get_summa(discounter => $discounter), "\n";
```

На выходе

```
196
210
```

Реализация

Класс Заказ

```
package Order;
sub new { return bless {}, shift }
sub get_summa {
    my $self = shift;
    my $opt = {@_};
    my $summa = $opt->{discounter}->do(summa => $self->{summa });
    return $summa;
}
1;
```

Фабрика `DiscountFactory` (ее кода здесь нет, там все как и в обычной фабрике) возвращает объекты класса либо `DiscountVisa`, либо

DiscountYM:

```
package DiscountVisa;
sub new { return bless {}, shift }

sub do {
    my $self = shift;
    my $opt  = {@_};

    # Здесь я позволила себе использовать
    # «магическое число» —— это только для наглядности
    # примера. Так делать плохо.
    return $opt->{summa} * (1 - 0.02);
}

package DiscountYM;
sub new { return bless {}, shift }

sub do {
    my $self = shift;
    my $opt  = {@_};
    return $opt->{summa} * (1 + 0.05);
}

1;
```

В классе `Order` у нас есть метод `get_summa`, который возвращает конечную стоимость заказа, но он должен учитывать и скидку на заказ. А скидка на заказ определяется способом оплаты заказа.

`my $discounter = DiscountFactory->new(type => 'Visa')` — создали наш объект-дискаунтер, который знает, как считать скидку при оплате картой Visa.

`$order->get_summa(discounter => $discounter)` — вызываем метод для получения итоговой стоимости заказа, передавая туда нашу «стратегию» расчета скидки.

`my $summa = $opt->{discounter}->do(summa => $self->{ summa });` — в методе `get_summa` мы вызываем операцию применения скидки к нашей базовой стоимости заказа.

То есть, мы передаем стратегию расчета скидки для заказа в качестве параметра. Эту же стратегию мы можем в дальнейшем исполь-

зовать и в других функциях, работающих со стоимостью заказа, заменять ее, не меняя остальной код.

На деле все очень просто, в следующей статье обязательно расскажу про другие очень используемые паттерны с чуть более сложной реализацией.

■ *Наталья Савенкова*

4. Perl 6 XXI века

Автор хочет дать еще один шанс шестому перлу

В октябре-ноябре на сайте конференции FOSDEM, которая пройдет в Брюсселе 31 января и 1 февраля 2015 года, появился анонс выступления Ларри Уолла, в котором сообщается, что будет объявлено, что Perl 6 станет готовым для продакшна в 2015 году.

Процитирую это полностью:

The last pieces are finally falling into place. After years of design and implementation, 2015 will be the year that Perl 6 officially launches for production use.

In this talk, the creator of Perl reflects on the history of the effort, how the team got some things right, and how it learned from its mistakes when it got them wrong. But mostly how a bunch of stubbornly fun-loving people outlasted the naysayers to accomplish the extraordinary task of implementing a language that was so ambitious, even its designers said it was impossible. Prepare to be delightfully surprised.

Из этого короткого сообщения совершенно непонятно, будет ли это сделано под Рождество (то есть через год в декабре), то ли прямо во время Фосдема. Мне кажется, что речь идет про декабрь и никак не про февраль, хотя некоторые комментаторы начали восторженно писать о том, что Perl 6 появится прямо в январе.

Чтобы посмотреть на текущее состояние Perl 6, надо начать с установки компилятора Rakudo, который, по сравнению с другими, развивается наиболее активно (что бы под этим не подразумевалось), и не исключено, что все-таки есть шанс, и мы сможем воспользоваться шестым перлом в обозримом будущем.

Установка с MoarVM

Краткая история развития Perl 6 включает в себя, помимо прочего, несколько эпизодов любви к виртуальным машинам, да и вообще вся история драматична. Первый тестовый компилятор был написан на C. Затем почти сразу появилась виртуальная машина Parrot, которая, однако, сперва хотела подмять под себя все языки на свете, а потом разработчики не справились с общением между собой, и проект остановился. Какое-то время, уже от других разработчиков, раздавались жалобы на то, что продолжать развитие с Parrot дальше невозможно: что-то там внутри не особо подходило для нужд Perl 6. Появился проект компилятора Rakudo с бекендом на виртуальной машине JVM (OMG!). А еще через какое-то время возник проект MoarVM, и компилятор переделан уже под нее.

Полная история намного богаче, тут и проект на Хаскеле PUGS, и не-совсем-перл NQP (Not Quite Perl) — упрощенная версия Perl 6, но достаточная для того, чтобы реализовать грамматики для компилятора языка, и стандартная грамматика STD.pm и еще много всего, что вспоминается как страшный сон.

Тем не менее, после YAPC::Europe 2013 последовал еще один рывок в разработке, и если попробовать то, что существует сегодня, окажется, что компилятор уже вполне быстрый, а таблица реализованных фич стала почти полностью зеленый. Мне еще раз хочется дать зеленый свет шестой версии перла, поэтому я и решил сдуть с него пыль и посмотреть, как обстоят дела сегодня.

Итого, на сегодня следует ориентироваться на компилятор Rakudo Star с бекендом MoarVM.

Установка тривиальна. Со страницы rakudo.org/downloads/star берется последний дистрибутив (сейчас это rakudo-star-2014.09.tar.gz), распаковывается и собирается вместе с нужной виртуальной машиной. В README указаны три варианта (для Parrot, JVM и MoarVM), но эти игры мы оставим разработчикам, а себе поставим свеженькое:

```
1 $ perl Configure.pl --backend=moar --gen-moar
2 $ make
3 $ make install
4 $ sudo cp perl6 /usr/bin
```

Эти действия, кроме последнего, выполняются от имени пользователя, последний шаг я сделал только для удобства (вместо этого вполне можно обойтись соответствующей правкой переменной `$PATH`).

Hello, World!

Как и для Perl 5, компилятор поддерживает и ключ командной строки `-e` (обратите внимание, что ключ `-E`, к которому приучил Perl 5 последних лет, здесь не нужен и не работает), и возможность прочитать программу из файла.

Вот такую, например:

```
say "Hello, Perl 6!";
```

Барабанная дробь:

```
$ perl6 hello.pl
Hello, Perl 6!
```

На моем тестовом компьютере время выполнения этой программы составило около 0,15 секунд. Это, конечно, безумно много для такой задачи, но в то же время это большое достижение по сравнению с тем, что было еще пару лет назад. Во всяком случае, теперь совсем неутомительно заново знакомиться с языком, потому что не приходится ждать долгой загрузки компилятора и собственно компиляции.

Любопытные могут познакомиться с ключом `--stagestats`, который расписывает время на выполнение основных этапов работы:

```
$ perl6 --stagestats hello.pl
Stage start      :   0.000
Stage parse      :   0.125
Stage syntaxcheck:   0.000
Stage ast        :   0.000
Stage optimize   :   0.001
Stage mast       :   0.006
Stage mbc        :   0.000
Stage moar       :   0.000
Hello, Perl 6
```

Как видно, в простейшей программе основное время уходит на разбор программы. Будет интересно вернуться к этой статистике попозже, когда мы разберем более сложный пример, с классами, например. Надо отметить, что стабильность компилятора и качество сообщений об ошибках тоже заметно возрасли.

Документация и набор тестов

За пятнадцать лет было написано очень много документации, которая неоднократно перерабатывалась и перекомпоновывалась. Документы, появившиеся вначале (и работа над которыми продолжалась), читать довольно тяжело, и для быстрого знакомства с языком они не очень удобны. Кроме того, многократно публиковались различные мануалы и начинались писаться книги, но и здесь надо быть осторожными, чтобы не напасть на устаревшее описание и неработающие примеры.

На сегодня знакомство следует начинать со страницы doc.perl6.org и, возможно, просмотреть большой комментированный пример на сайте Learn X in Y minutes. Ссылки на остальные документы находятся на странице perl6.org/documentation — чем выше в списке ссылка, тем больше вероятность, что документ не слишком устарел.

Отдельным источником знаний, как и сто лет назад, могут служить примеры из набора тестов, доступных в дистрибутиве в каталоге `rakudo/t/spec`. Примеров очень много, они сгруппированы по темам в соответствии с номерами и разделами основополагающих документов *Synopses*. Дополнительно следует посмотреть на репозиторий github.com/perl6/perl6-examples.

Переменные

В Perl 6 для переменных используются сигилы, частично совпадающие с тем, что есть в Perl 5. В частности, скаляры, списки и хеши используют, соответственно, сигилы `$`, `@` и `%`.

```
1 my $scalar = 42;
```

```
2 say $scalar;
```

Никакого сюрприза, напечатается 42.

```
1 my @list = (10, 20, 30);
2 say @list;
```

Здесь тоже все очевидно и предсказуемо:

```
1 10 20 30
```

Однако, сразу можно воспользоваться преимуществами синтаксиса Perl 6 и записать те же инструкции, используя меньшее число символов и пунктуации:

```
1 my @list1 = <10 20 30>;
```

Или даже так (вообще кайф):

```
1 my @list2 = 10, 20, 30;
```

Точно так же при инициализации хеша допустимо опустить скобки, оставив только контент:

```
1 my %hash = 'Language' => 'Perl', 'Version' => '6';
2 say %hash;
```

На выводе появится следующее:

```
1 "Language" => "Perl", "Version" => "6"
```

Доступ к элементам списка и хеша осуществляется с помощью знакомых скобок, но при этом сигил не меняется. В следующих примерах из списка или хеша извлекается скалярная величина:

```
1 my @squares = 0, 1, 4, 9, 14, 25;
2 say @squares[3]; # выводит четвертый элемент, то есть 9
3
4 my %capitals = 'France' => 'Paris', 'Germany' => 'Berlin'
    , 'Ukraine' => 'Kiev';
5 say %capitals{'Ukraine'};
```

Существует альтернативный синтаксис как для создания хеша, так и для доступа к его элементам. Как это происходит, видно из таких

примеров (допустимо смешивать любые стили объявления и доступа):

```
1 my %length-abbrs = :m('meter'), :km('kilometer'), :cm('
    centimeter');
2 say %length-abbrs<km>; # выводится kilometer
```

В именах переменных разрешено использовать не только буквы, цифры и символ подчеркивания, но и, например, дефис, апостроф и юникод:

```
1 my $hello-world = "Hello, World";
2 say $hello-world;
3
4 my $don't = "Порошок, уходи!";
5 say $don't;
6
7 my $привет = "Привет всем!";
8 say $привет;
```

(Несмотря на возможную пользу, набирать на клавиатуре это неудобно, потому что в пределах одного имени приходится переключаться между русской и английской раскладками клавиатуры.)

Интроспекция

В Perl 6 встроен механизм, позволяющий очень просто узнать тип данных, хранящихся в контейнере. Для этого используется метод `.WHAT`, который вызывается непосредственно на интересующей переменной. Для переменных, начинающихся с сигилов `@` и `%`, значениями будут `(Array)` и `(Hash)`, а для скаляров (`$`) результат интроспекции будет зависеть от данных, фактических находящихся в переменной:

```
1 say $scalar.WHAT;
2 say $hello-world.WHAT;
3 say $привет.WHAT;
```

Эти три строки напечатают такие три ответа (вместе со скобками):

```
1 (Int)
2 (Str)
3 (Str)
```

Соответственно, для массивов (опять, здравствуй, путаница между списками и массивами!) и хешей:

```
say @list.WHAT;
say @squares.WHAT;
```

Результат:

```
(Array)
(Array)
```

Теперь с хешами:

```
say %hash.WHAT;
say %capitals.WHAT;
```

Предсказуемо напечатается:

```
(Hash)
(Hash)
```

Можно пойти дальше и вывести имя переменной:

```
say $scalar.VAR.name;
```

Напечатается:

```
$scalar
```

То, что возвращается методом `.WHAT`, является так называемым объектом типа (type object). В язык встроен оператор ===, предназначенный для сравнения таких объектов типа. Например:

```
my $value = 42;
say "OK" if $value.WHAT === Int;
```

Альтернатива — метод `.isa`, вызванный на объекте.

```
say "OK" if $value.isa(Int);
```

Твигилы

В Perl 6 перед именем переменной может стоять как один сигил (символ `$`, `@` или `%`), так и два. Второй символ, называемый твигилом

(twigil), может указывать, например, на изменение области видимости переменной.

Например, `*` указывает на динамическую область видимости. В частном случае это означает просто глобальную переменную. Вот пример программы, которая построчно выводит аргументы командной строки.

```
.say @*ARGS;
```

Здесь массив `@*ARGS` — глобальный массив с аргументами командной строки (называется не `ARGV`, а именно `ARGS`). Конструкция `.say` — вызов метода `.say()` для переменной цикла; более развернуто это можно было бы записать так:

```
for @*ARGS {
    $_.say;
}
```

Еще несколько полезных предопределенных динамических переменных со звездочкой. Первый сигил, как и прежде, обозначает тип контейнера (скаляр, массив или хеш):

- `$*PERL` содержит версию перла (`Perl 6`);
- `$*PID` — номер процесса;
- `$*PROGRAM_NAME` — имя файла с программой, которая сейчас исполняется (для однострочников внутри `-e` переменная содержит строку `-e`);
- `$*EXECUTABLE` — путь к интерпретатору;
- `$*VM` содержит название виртуальной машины, с которой скомпилирован perl6;
- `$*DISTRO` — название и версия дистрибутива операционной системы;
- `$*KERNEL` — аналогично, но для версии ядра;
- `$*CWD` — текущий рабочий каталог;
- `$*TZ` — текущая временная зона;
- `@*INC` — нечто, похожее на список каталогов для поиска модулей;
- `%*ENV` — переменные окружения.

В моем случае значения скалярных переменных из этого списка оказались такими:

```
1 Perl 6
2 1190
3 globals.pl
4 IO::Path</usr/bin/perl6>
5 moar (2014.9)
6 linux (2.6.32.5.amd.64)
7 linux (1.SMP.Mon.Feb.25.0.26.11.UTC.2013)
8 IO::Path</home/ash/perl6/test>
9 3600
```

Стоит обратить внимание на то, что пути к файлам указаны как `IO::Path<...>`, а в переменной `$*TZ` содержится смещение от UTC в секундах.

Следующий блок имен — с твигилом `?`. Это «константы» (compile-time "constants"), помогающие понять, в каком месте программы мы сейчас находимся.

- `$?FILE` — имя файла с программой (без пути; содержит строку –e, если вся программа находится внутри одноименного ключа);
- `$?LINE` — номер строки (1 для однострочников);
- `$?PACKAGE` — имя текущего модуля, на верхнем уровне это `(GLOBAL)`;
- `$?TABSTOP` — число пробелов в табуляции (по-видимому, может пригодиться в heredoc-ах).

Частоиспользуемые специальные переменные

Переменная `$_` служит точно тем же целям, что и в Perl 5. При этом стоит иметь в виду, что в Perl 6 она может являться объектом даже в самых простых случаях. Например, недавний пример с печатью аргументов командной строки содержал `$_.say`. То же самое допустимо записать в виде `$_.say()` или просто `.say()` или `.say`.

Эта же переменная используется по умолчанию в некоторых других местах, например, при сопоставлении с регулярным выражением:

```
1 for @*ARGS {
2     .say if /\d/;
3 }
```

Полная запись выглядела бы так (используется оператор «умного сравнения» ~~ (smart match)):

```
1 for @*ARGS {
2     $_.say if $_ ~~ /\d/;
3 }
```

Результат сопоставления с регулярным выражением доступен в переменной `$/`. Чтобы получить совпавшую строку, достаточно вызвать метод `$/.Str`, а для доступа к захваченным подстрокам — обратиться по индексу: `$/[2]` (или просто написать `$2`).

```
1 "Birthday: 18 December 2014" ~~ /(\d+)\s(\D+)\s(\d+)/;
2 say $/.Str;
3 say $/[$_] for 0..2;
```

В этой программе в строке будет найдена дата (последовательность из цифр, пробела, слова из не-цифр, пробела и еще немного цифр). После успешного сопоставления вызов `$/.Str` содержит дату, а `$/[0]` — `$/[2]` ее отдельные части (японские скобки — часть вывода):

```
1 18 December 2014
2 18
3 December
4 2014
```

Наконец, переменная `$!` содержит сообщение об ошибке, возникшей внутри блока `try` или, например, при открытии файла.

```
1 try {
2     say 42/0;
3 }
4 say $!;
```

Если убрать последнюю строку `say $!`, то программа завершится, ничего не напечатав. Но в этом примере будет выведено и сообщение об ошибке (точно такое же, которое бы возникло при отсутствии `try`).

Встроенные типы

В Perl 6 без дополнительных сложностей возможно использовать типизированные переменные, указав при объявлении переменных один из встроенных типов.

Часть типов очевидна и не требует пояснений:

- `Bool`
- `Int`
- `Str`
- `Array`
- `Hash`
- `Complex`

Про другие следует сказать пару слов:

- `Num`
- `Pair`

Тип `Num` предназначен для чисел с плавающей точкой, а `Pair` — пара объектов «ключ — значение».

Типизированные переменные

При объявлении переменной тип указывают таким образом:

```
1 my Int $x;
```

В этом случае скалярный контейнер сможет содержать только целые числа, и попытка записать туда что-то другое приведет к ошибке:

```
1 my Int $x;
2 $x = "abc"; # Ошибка: Type check failed in assignment to
    '$x';
3             #         expected 'Int' but got 'Str'
```

Для преобразования типов следует воспользоваться одноименным методом, вызванном на объекте другого типа, например:

```
1 my Int $x;
2 $x = "123".Int; # Теперь OK
3 say $x; # 123
```

Bool

Использование переменных типа `Bool` довольно очевидно, но есть особенности, на которые интересно обратить внимание. Тип `Bool` является встроенным перечислением (built-in enumeration) и предоставляет программисту два значения: `True` и `False` (в полной записи: `Bool::True`, `Bool::False`). Переменные этого типа можно инкрементировать или декрементировать, например:

```
1 my $b = Bool::True;
2 $b——;
3 say $b; # выведется False
4
5 $b = Bool::False;
6 $b++;
7 say $b; # True
```

Кроме того, существует метод `.Bool`, который можно вызывать на объектах других типов, например:

```
1 say 42.Bool; # True
2
3 my $pi = 3.14;
4 say $pi.Bool; # True
5
6 say 0.Bool; # False
7 say "00".Bool; #True
```

Аналогично, можно вызвать метод `.Int` и получить целочисленное представление булевых (и любых других) значений:

```
1 say Bool::True.Int; # 1
```

Int

Тип `Int` предназначен для хранения целых чисел произвольного размера. Например, в этом примере ничего не теряется:

```
my Int $x = 12389147319583948275874801735817503285431532;
say $x;
```

Для записи чисел с основой, отличающейся от 10, существует такой синтаксис:

```
say :16<D0CF11E0>
```

По-прежнему разрешается использовать символ подчеркивания для облегчения чтения длинных чисел:

```
my Int $x = 735_817_503_285_431_532;
```

На объекте типа `Int` можно вызвать интересные методы, например, для преобразования в символ или (экзотика!) для проверки, является ли число простым:

```
my Int $a = 65;
say $a.chr; # A

my Int $i = 17;
say $i.is-prime; # True

say 42.is-prime; # False
```

Str

`Str` это, разумеется, строки. В Perl 6 многие методы для работы со строками являются именно методами, которые вызывают на строке как на объекте:

```
my $str = "My string";

say $str.lc; # my string
say $str.uc; # MY STRING

say $str.index('t'); # 4
```

Теперь попробуем узнать длину строки. Первая наивная попытка вызвать метод `$str.length` заканчивается неудачей, но при этом с полезной подсказкой:

```
No such method 'length' for invocant of type 'Str'
Did you mean 'elems', 'chars', 'graphs' or 'codes'?
```

То есть появился удобный и однозначный способ определить длину юникодной строки (и непонятно куда пропал простой способ подсчитать байты):

```
say "Перл 6".chars; # 6
```

При работе со строками придется какое-то время привыкать к новому подходу:

```
"Today is %02i %s %i\n".printf($day, $month, $year);
```

Array

У переменных типа `Array` (то есть у всех переменных, начинающихся сигилом `@`) есть пара простых методов, которые могут оказаться полезными:

```
my @a = 1, 2, 3, 5, 7, 11;
say @a.Int; # длина массива
say @a.Str; # значения, разделенные пробелом
```

Hash

Для хешей предусмотрено несколько понятных методов, например, таких:

```
say %hash.elems;  # число пар в хеше
say %hash.keys;   # список ключей
say %hash.values; # список значений
```

Возможно получить не только отдельные ключи или значения, но и сразу пары элементов:

```
1 for %hash.pairs {
2     say $_.key;
3     say $_.value;
4 }
```

С помощью метода `.invert` возможно получить список пар, в которых ключ и значения поменяны местами:

```
1 for %hash.invert {
2     .key.say; # то же, что и say $_.key
3     .value.say;
4 }
```

Наконец, метод `.kv` возвращает список, состоящий из чередующихся ключей и значений элементов хеша:

```
1 say %hash.kv
```

Функции

Объявление функции без аргументов и ее вызов выглядят знакомо и очень просты:

```
1 sub call-me {
2     say "I'm called"
3 }
4
5 call-me;
```

Объявление аргументов функции сделано аналогично тому, как это выглядит в других языках (в том числе в Perl 5.20):

```
1 sub cube($x) {
2     return $x ** 3;
3 }
4
5 say cube(3); # 27
```

Обязательные аргументы указываются в скобках через запятую, както дополнительно их объявлять не требуется:

```
1 sub min($x, $y) {
2     return $x < $y ?? $x !! $y;
```

```
3 }
4
5 say min(-2, 2);
6 say min(42, 24);
```

(Тернарный оператор в Perl 6 выглядит как `??` ... `!!`.)

Объявленные таким образом аргументы являются обязательными, и вызов функции с другим числом параметров приведет к ошибке.

Передача не по значению

Аргументы передаются по значению, и более того, внутри функции их изменить не получится. Чтобы передать аргументы по ссылке (хотя формально это называется не передачей по ссылке, а передачей изменяемой (mutable) переменной), достаточно указать свойство `is rw`:

```
1 sub inc($x is rw) {
2     $x++;
3     return $x;
4 }
5
6 my $value = 42;
7 inc($value);
8 say $value; # 43
```

Типизированные параметры

Аналогично тому, как указывался тип при объявлении переменных, возможно сообщать компилятору о том, аргументы каких типов ожидает функция:

```
1 sub say-hi(Str $name) {
2     say "Hi, $name!";
3 }
```

Если типы совпадают, все ОК, а если нет, возникает ошибка (причем на этапе компиляции).

```
say-hi("Mr. X"); # Допустимо

#say-hi(123); # Calling 'say-hi' will never work with argument types (int)
              # Expected: :(Str $name)
```

Необязательные параметры

Необязательность аргументов функции обозначается вопросительным знаком после имени. Проверку того, что аргумент передан, можно выполнить, вызвав функцию defined:

```
sub send-mail(Str $to, Str $bcc?) {
    if defined $bcc {
        # . . .
        say "Sent to $to with a blind carbon copy to $bcc.";
    }
    else {
        # . . .
        say "Sent to $to.";
    }
}

send-mail('mail@example.com');

send-mail('mail@example.com', 'larry@wall.org');
```

Значения по умолчанию

В Perl 6 предусмотрен и механизм для указания значения аргументов функции по умолчанию. Синтаксически это выглядит таким образом:

```
sub i-live-in(Str $city = "Moscow") {
    say "I live in $city.";
}

i-live-in('Saint Petersburg');

i-live-in();
```

Помимо константных значений, известных на момент компиляции, возможно вычислять значения по умолчанию во время выполнения, явно указав вызов функции после знака =:

```
sub to-pay($salary, $bonus = 100.rand) {
    return ($salary + $bonus).floor;
}

say to-pay(500, 50); # Всегда на руки 550.
say to-pay(500); # Может быть что угодно от 500 до 600.
say to-pay(500); # Тот же вызов, но скорее всего другой результат.
```

Еще раз обратите внимание на то, что `.rand` и `.floor` вызываются как методы, а не как функции.

Аргументы, которые необязательны или содержат значения по умолчанию, должны следовать после всех обязательных (иначе компилятор не сможет понять, какие параметры переданы).

Именованные аргументы

Помимо позиционных аргументов (то есть тех, которые при вызове функции следует передавать в том же порядке, в котором они объявлены), возможно передавать параметры по именам, примерно в том же стиле, как это делают в Perl 5, передавая параметры в хеше. Чтобы сообщить об именованном параметре, достаточно поставить перед ним двоеточие:

```
sub power(:$base, :$exponent) {
    return $base ** $exponent;
}
```

Теперь можно передавать параметры в любом порядке, результат от этого не изменится:

```
say power(:base(2), :exponent(3)); # 8
say power(:exponent(3), :base(2)); # 8
```

Если хочется использовать разные имена для переменных внутри функции и для аргументов, то надо указать это имя таким образом:

```
sub power(:val($base), :pow($exponent)) {
    return $base ** $exponent;
}
```

Теперь при вызове ожидаются новые имена:

```
say power(:val(5), :pow(2)); # 25
say power(:pow(2), :val(5)); # 25
```

Сворачивание и разворачивание

В функциях Perl 6 реально в любом удобном порядке смешивать скаляры и списки. Массив может оказаться в списке аргументов на первом месте, а после него может идти скаляр. В следующем примере список `@text` доступен внутри функции, и он содержит ровно те значения, которые были переданы извне.

```
sub cute-output(@text, $before, $after) {
    say $before ~ $_ ~ $after for @text;
}

my @text = <C C++ Perl Go>;
cute-output(@text, '{', '}');
```

На выходе появится ожидаемая красота:

```
{C}
{C++}
{Perl}
{Go}
```

Язык ожидает, что функция получит аргументы именно тех типов, которые указаны в ее объявлении. Поэтому, например, если функция объявлена с одним аргументом-списком, она не сможет принять произвольное число скаляров.

```
sub get-array(@a) {
    say @a;
}

get-array(1, 2, 3); # Ошибка: Calling 'get-array' will
                    # never work with argument types (Int
                       , Int, Int)
```

Для такого поведения следует явно указать, что аргумент функции является *slurpy*, поставив перед ним звездочку:

```
1 sub get-array(*@a) {
2     say @a;
3 }
4
5 get-array(1, 2, 3); # Все ОК: 1 2 3
```

Аналогично, не будет работать и в обратную сторону, когда функция ожидает несколько скаляров, а при вызове получает массив:

```
1 sub get-scalars($a, $b, $c) {
2     say "$a and $b and $c";
3 }
4
5 my @a = <3 4 5>;
6 get-scalars(@a); # Ошибка: Calling 'get-scalars' will
7                  # never work with argument types (
                      Positional)
```

Чтобы развернуть массив в последовательность скаляров, надо поставить перед ним вертикальную черту:

```
1 get-scalars(|@a); # 3 and 4 and 5
```

Еще немного про функции

Допустимо создавать вложенные функции:

```
1 sub cube($x) {
2     sub square($x) {
3         return $x * $x;
4     }
5
6     return $x * square($x);
7 }
8
9 say cube(3); # 27
```

При этом вложенная функция `square` видна только внутри тела `cube`
.

Интересно посмотреть на создание анонимных функций. Один из вариантов (а их несколько) синтаксических правил выглядит так (чем-то напоминает типовые конструкции в jQuery):

```
1 say sub ($x, $y) {$x ~ ' ' ~ $y}("Perl", 6);
```

Здесь первые круглые скобки содержат список формальных аргументов анонимной функции, вторые круглые скобки содержат переданные ей значения, а тело функции находится в фигурных скобках. Важно, кстати, чтобы после закрывающей фигурной скобки не было пробела (зачем так?!). Примечание: оператор конкатенации строк в Perl 6 — ~.

Классы

С объектно-ориентированной направленностью Perl 6 мы уже неоднократно встретились, когда вызывали методы на переменных, которые на первый взгляд не являются объектами, а в некоторых случаях методы вызывались не на переменных, а на константах. В этих случаях объектно-ориентированное поведение мало заметно в коде, поскольку оно скрыто внутри компилятора.

В языке существуют два вида типов: обычные и нативные. Нативные типы — то, что поддерживается непосредственно оборудованием (то есть `int`, `uint32` и т. п.). А то, что мы видели ранее (например, `Int` или `Str`) — это типы-контейнеры, которые содержат переменные соответствующих нативных типов. Компилятор самостоятельно выполняет нужные преобразования, если это требуется для работы программы. Например, когда происходит вызов `42.say`, то вызывается метод `.say`, определенный для объектов типа `Int`, который в свою очередь наследуется от типа `Mu`, стоящего на вершине иерархии классов в Perl 6.

Что же касается ООП в традиционном понимании, то в Perl 6 это сделано совершенно иначе, чем в Perl 5. Синтаксис более прозрачен и ближе к тому, что встречается в других языках с классами:

```
1 class Cafe {
2 }
```

Данные класса

Члены-данные класса объявляют с помощью ключевого слова `has`, а область видимости определяется твигилом: точка — поле доступно публично (через автоматически генерируемые аксессоры), восклицательный знак — поле приватно.

```
class Cafe {
    has $.name;
    has @!orders;
}
```

Чтобы создать объект класса `X`, требуется вызвать конструктор `X.new()` — этот метод неявно унаследован от класса `Mu`:

```
my $cafe = Cafe.new(
    name => "Paris"
);
```

Теперь возможно читать публичные поля:

```
say $cafe.name;
```

Однако, чтобы изменить значение поля извне, необходимо явно его указать с атрибутом, разрешающим чтение и запись:

```
class Cafe {
    has $.name is rw;
    has @!orders;
}

my $cafe = Cafe.new(
    name => "Paris"
);

$cafe.name = "Berlin";
say $cafe.name;
```

Методы класса

Для создания метода класса предусмотрено ключевое слово `method`, а в остальном метод похож на обычную функцию, которая, разумеется, может обращаться к любым данным класса, как публичным,

так и к приватным. Метод тоже может быть приватным, для этого достаточно поставить перед его именем восклицательный знак (вернемся к этому после наследования).

В этом коротком примере создано два метода, которые оперируют массивом `@!orders`:

```
class Cafe {
    has $.name;
    has @!orders;

    method order($what) {
        @!orders.push($what);
    }

    method list-orders {
        @!orders.sort.join(', ').say;
    }
}

my $cafe = Cafe.new(
    name => "Paris"
);

$cafe.order('meet');
$cafe.order('fish');
$cafe.list-orders; # fish, meet
```

Как видно, код довольно понятен для тех, кто знаком с ООП. Отдельно еще раз обращу внимание на то, как на практике проявляется факт того, что всё — объект:

```
@!orders.sort.join(', ').say;
```

Внутри методов доступна указывающая на текущий объект переменная `self`, через которую можно обращаться к данным экземпляра или к методам класса:

```
method order($what) {
    @!orders.push($what);
    self.list-orders;
}

method list-orders {
    say self.name;
    @!orders.sort.join(', ').say;
}
```

Наследование

Наследование реализовать крайне просто: при объявлении класса достаточно указать имя базового с помощью ключевого слова `is`:

```
class A {
    method x {
        say "A.x"
    }
    method y {
        say "A.y"
    }
}

class B is A {
    method x {
        say "B.x"
    }
}
```

Дальше особо объяснять ничего не требуется:

```
my $a = A.new;
$a.x; # A.x
$a.y; # A.y

my $b = B.new;
$b.x; # B.x
$b.y; # A.y
```

Важно, что результат поиска метода не зависит от того, какой тип из иерархии был указан при объявлении переменной. Perl 6 всегда исходит из того, какой объект фактически находится в контейнере. Поэтому, например, если в предыдущем примере объявить переменную `$b` типа `A`, то вызов `$b.x` по-прежнему попадет в метод дочернего класса:

```
my A $b = B.new;
$b.x; # B.x
$b.y; # A.y
```

Увидеть точный порядок, в котором происходит разрешение методов, позволяет спецметод `.^mro`:

```
say $b.^mro;
```

В этом примере на печати появится:

```
(B) (A) (Any) (Mu)
```

Кстати, `.^mro` можно вызвать и на любом другом объекте в программе, чтобы краем глаза посмотреть на внутреннюю реализацию:

```
$ perl6 –e'42.^mro.say'
(Int) (Cool) (Any) (Mu)
```

Множественное наследование

Множественное наследование получают, перечисляя все нужные классы:

```
class A {
    method a {
        say "A.a"
    }
}

class B {
    method b {
        say "B.b";
    }
}

class C is A is B {
}

my $c = C.new;
$c.a;
$c.b;
```

При конфликте имен порядок перечисления родителей в объявлении класса имеет значение:

```
class A {
    method meth {
        say "A.meth"
    }
}

class B {
    method meth {
```

```
9           say "B.meth";
10      }
11  }
12
13  class C is A is B {
14  }
15
16  class D is B is A {
17  }
```

В этом примере метод с именем `.meth` существует в обоих родительских классах, поэтому будучи вызванным на переменных типа `C` или `D`, он приведет к разным методам:

```
1 my $c = C.new;
2 $c.meth; # A.meth
3
4 my $d = D.new;
5 $d.meth; # B.meth
```

Порядок разрешения имен подтверждает это:

```
1 $c.^mro.say; # (C) (A) (B) (Any) (Mu)
2 $d.^mro.say; # (D) (B) (A) (Any) (Mu)
```

Приватные (закрытые) методы

После того, как расмотрено наследование, можно вернуться к приватным или закрытым методам. Такие методы разрешается вызывать только в пределах текущего класса. Они недоступны ни извне, ни в дочерних классах. И объявление, и использование содержит восклицательный знак:

```
1  class A {
2      # Метод доступен только внутри A
3      method !private {
4          say "A.private";
5      }
6
7      # Открытый метод, который обращается к закрытому
8      method public {
9          # Без self не получится, а ! используется как
              точка
10         self!private;
```

```
11     }
12 }
13
14 class B is A {
15     method method {
16         # Здесь тоже self, но уже с точкой, потому что
               метод публичный
17         self.public;
18
19         # А это приведет к ошибке компиляции
20         #self!private;
21     }
22 }
23
24 my $b = B.new;
25 $b.method; # A.private
```

Подметоды

В Perl 6 существует понятие подметодов — это такие методы класса, которые доступны только в пределах текущего класса (при этом они могут быть публичными), но не наследуются потомками. Вот пример, в котором создается дочерний класс, но подметод из родительского класса там не просто недоступен — он там отсутствует:

```
1 class A {
2     submethod submeth {
3         say "A.submeth"
4     }
5 }
6
7 class B is A {
8 }
9
10 my A $a;
11 my B $b;
12
13 $a.submeth;  # OK
14 #$b.submeth; # Не OK
```

Конструкторы

Внимательный читатель должно быть заметил разные способы создания переменных в предыдущих примерах.

С явным вызовом метода `.new` (создается объект):

```
1 my $a = A.new;
```

или просто с объявлением типа переменной (создается контейнер):

```
1 my A $a;
```

И то, и другое уживается вместе (создаются и объект, и контейнер для него):

```
1 my A $a = A.new;
```

Различие становится очевидным, если учесть, что методы класса уже определены в коде, а для доступа к данным объекта требуется собственно сам объект. Рассмотрим это на примере класса, в котором есть один публичный метод и одно публичное поле:

```
1 class A {
2     has $.x = 42;
3     method m {
4         say "A.m";
5     }
6 }
```

Здесь же показан способ инициализации переменных с данными объекта.

Теперь создадим скалярный контейнер класса `A`:

```
1 my A $a;
```

Контейнер создался, его тип известен, но данных еще нет. Поэтому, метод может быть вызван:

```
1 $a.m; # Печатает "A.m"
```

а поле `$.x` еще недоступно:

```
1 say $a.x; # Ошибка: Cannot look up attributes in a type
    object
2           #         in method x
```

Поэтому необходимо создать инстанс, вызвав конструктор, после чего все работает:

```
1 my A $b = A.new;
2 say $b.x; # Выводится 42
```

Важно отметить, что несмотря на то, что в определении класса был инициализатор поля (`= 42`), само поле создается только после вызова `.new`.

Предопределенный метод `.new`, унаследованный от класса `Mu`, принимает список именованных аргументов. Соответственно, этот метод удастся вызвать на объекте любого класса и в нем передать ему требуемые значения полей:

```
1 my A $c = A.new(x => 14);
2 say $c.x; # 14, а не 42
```

Примечание: заключать в кавычки имя переменных (например, `A.new('x' => 14)`) не нужно, это приведет к ошибке.

Для реализации более сложных конструкторов, которые подразумевают не только копирование данных, следует создать подметод `BUILD`. Ожидается, что этот метод принимает список именованных аргументов.

```
1 class A {
2     # В объекте два поля, одно из которых будет
         вычисляться в конструкторе.
3     has $.str;
4     has $!len;
5 
6     # Конструктор ожидает один аргумент с именем str.
7     submethod BUILD(:$str) {
8         # Одно поле копируется как есть:
9         $!str = $str;
10 
11        # А второе вычисляется:
12        $!len = $str.chars;
13    }
14 
```

```
15     method dump {
16         # Здесь просто выводим текущие значения.
17         # Переменные интерполируются как обычно, но чтобы
               апостроф
18         # не попал в имя переменной, стоят фигурные
              скобки.
19         "{$.str}'s length is $!len.".say;
20     }
21 }
22
23 my $a = A.new(str => "Perl");
24 $a.dump;
```

Эта программа напечатает строку `Perl's length is 4.`

О доступе к данным

В документации рекомендуется внутри класса всегда использовать конструкцию с восклицательным знаком независимо от того, публичное это поле или нет. Предполагается, что обращение `$.str` должно быть реализовано через вызов метода, а `$!str` — прямым доступом к переменной.

Иными словами, запись `$.x` является сокращением для конструкции, создающей публичный одноименный метод для чтения значения приватной переменной:

```
1 $!x;
2 method x() {
3     $!x
4 }
```

Об этой особенности важно помнить, если потребуется изменять значения (собственно, компилятор об этом напомнит). С практической точки зрения, внутри класса проще всегда использовать восклицательный знак.

Следующий пример работоспособен, но закомментированная строка его сломала бы:

```
1 class A {
2     has $.x;
```

```
3
4     method change($value) {
5         #$.x = $value; # Ошибка: Cannot modify an
             immutable Int
6         $!x = $value;
7     }
8 }
9
10 my $a = A.new(x => 2);
11 $a.change(7);
12 $a.x.say; # 7
```

Другой вариант, который уже встречался ранее, — использование атрибута `is rw`.

Роли

Рядом с классами в Perl 6 существуют роли. Это то, что в других языках называют интерфейсами. Методы и данные, определенные в роли, затем можно добавить к новому классу через наследование, используя слово `does`. Роль — это по сути класс, методы и данные которого при наследовании становятся частью класса (а не наследуются как при наследовании классов). Поэтому при конфликте имен об этом станет известно уже на этапе компиляции, и вычислять порядок обхода классов для поиска нужно имени не потребуется.

Следующий пример описывает роль, которая затем используется при создании двух классов. В этом примере тот же результат мог быть достигнут обычным наследованием классов. Читателю предлагается придумать более изощренный пример, где польза ролей станет более очевидной.

```
1 # Роль пункта питания — можно принимать заказы (метод
     order),
2 # подсчитывать сумму заказа (метод calc) и выставлять
     счет (метод bill).
3 role FoodService {
4     has @!orders;
5
6     method order($price) {
7         @!orders.push($price);
8     }
```

```
9
10      method calc {
11          # [+] это гипероператор, которым связываются все
               элементы массива.
12          # То есть запись [+] @a равнозначна @a[0] + @a[1]
                + ... + @a[N].
13          return [+] @!orders;
14      }
15
16      method bill {
17          # Сумма счета пока ничем не отличается от суммы
               заказов.
18          return self.calc;
19      }
20  }
21
22  # Строим кафе. Кафе — это пункт питания.
23  class Cafe does FoodService {
24      method bill {
25          # Но с небольшой наценкой.
26          return self.calc * 1.1;
27      }
28  }
29
30  # Открываем ресторан
31  class Restaurant does FoodService {
32      method bill {
33          # Сначала парим клиента некоторое время.
34          sleep 10.rand;
35
36          # А потом еще делаем ресторанную наценку.
37          return self.calc * 1.3;
38      }
39  }
```

Проверяем все в действии. Сначала кафе:

```
1 my $cafe = Cafe.new;
2 $cafe.order(10);
3 $cafe.order(20);
4 say $cafe.bill; # Сразу 33
```

Затем ресторан (задержка с ответом вызвана не скоростью Perl 6, а сутью ресторана):

```
1 my $restaurant = Restaurant.new;
2 $restaurant.order(100);
3 $restaurant.order(200);
```

```
4 say $restaurant.bill; # 390 неизвестно когда
```

Продолжение, возможно, следует

На этом пока все, однако описанное в этом номере журнала покрывает лишь небольшую часть того, что входит в Perl 6. Где-то остались неосвещенными детали, где-то можно продолжить большими списками новых операторов, описаниями встроенных типов для работы со множествами и т. д. Кроме того, пара тем слишком объемна для первого раза: это регулярные выражения (они новые) и грамматики. И, наконец, не менее интересна тема, связанная с параллельными вычислениями.

■ *Андрей Шитов*

5. DBIx::Class. Сборник рецептов

Сборник рецептов на все случаи жизни

В данной статье собраны рецепты использования DBIx::Class по следующим темам:

- создание простых запросов;
- выборка строк с помощью where, distinct, group by, having;
- выборка строк по первичному и уникальному ключам;
- использование custom-методов для Result- и ResultSet-классов;
- использование отношений между таблицами;
- подзапросы;
- ограничение результатов поиска с помощью limit;
- CRUD для строк;
- CRUD;
- CRUD с поиском.

Для начала создадим БД для работы. Это будет информация о некой компании, ее служащих и об отделах, в которых они работают.

Загрузить SQL-файл с кодом создания и добавления данных, а также схему БД с подробным описанием отношений между таблицами в Result-классе каждой таблицы можно по ссылке.

Простые запросы

Рецепт 1. Получить список resultset-объектов для всей выборки с помощью метода `all`

Решение:

```
1 say $_->name for $department_rs->all;
```

Для получения столбца мы вызываем метод-аксессор с именем столбца.

Рецепт 2. Итерируем выборку с помощью метода `next`

Решение:

```
1 while (my $department = $department_rs->next) {
2   say $department->name;
3 }
```

Данный вариант может быть использован для эффективного итерирования каждой записи в выборке (`resultset`-е).

Рецепт 3. Итерируем выборку с помощью курсора

Решение:

```
1 my $client_cursor = $client_rs->cursor;
2 while (my @client_row = $client_cursor->next) {
3   say $client_row[1]; # client name
4 }
```

Для итерирования используется метод `next` класса `DBIx::Class::Cursor`, который возвращает следующую строку из курсора в виде массива значений столбцов (результат работы метода `fetchrow_array` из DBI), т.е. на каждой итерации получаем следующую структуру:

```
1 [
2     [0] 1,
3     [1] "Telco Inc",
4     [2] "1 Collins St Melbourne",
5     [3] "Fred Smith",
6     [4] 95551234
7 ]
```

Рецепт 4. Получить массив хешей со значениями таблицы

Решение:

```
1 my @department = $department_rs->search(undef,
2     {result_class => 'DBIx::Class::ResultClass::
         HashRefInflator'})->all;
3 say 'Department name: ' . $department[0]{name};
```

На выходе получается следующая структура:

```
1 [
2     [0] {
3         departmentid   42,
4         name           "Финансовый отдел"
5     },
6     [1] {
7         departmentid   128,
8         name           "Отдел проектировия"
9     },
10    ...
11 ]
```

Рецепт 5. Итерируем набор хешей со значениями таблицы

Решение:

```
1 my $rs = $department_rs->search(undef,
2     {result_class => 'DBIx::Class::ResultClass::
         HashRefInflator'});
3 while (my $hashref = $rs->next) {
4     say $hashref->{name};
5 }
```

На каждой итерации получаем ссылку на хеш такого вида:

```
1 \ {
2     departmentid   42,
3     name           "Финансовый отдел"
4 }
```

Рецепт 6. Посчитать количество строк в таблице

Запрос, который мы хотим получить:

```
1 select count(departmentID) from department;
```

Решение:

```
1 my $department_cnt =
2   $department_rs->get_column('departmentid')->func('count
     ');
3 say "departmentid count = $department_cnt";
```

Здесь мы с помощью метода `get_column()` получаем объект класса `DBIx::Class::ResultSetColumn`, который позволяет работать с отдельными столбцами из результирующей выборки. Далее мы применили метод `func` из этого класса, который принимает имя SQL-функции и добавляет ее в select-запрос для выбранного столбца, в нашем случае departmentID.

Если нужно реализовать запрос вида:

```
1 select count(*) from department;
```

то можно написать следующим образом:

```
1 my $cnt = $department_rs->count;
2 say "department count = $cnt";
```

Рецепт 7. Посчитать максимальное значение departmentID

Запрос, который мы хотим получить:

```
1 select max(departmentID) from department;
```

Решение:

```
1 my $max = $department_rs->get_column('departmentid')->max
    ;
2 say "department max = $max";
```

Как и в прошлом рецепте, здесь используется метод max из класса DBIx::Class::ResultSetColumn, который подсчитывает максимальное значение заданного столбца. Метод max является алиасом для func('max').

Рецепт 8. Выбор отдельных столбцов

Запрос, который мы хотим получить:

```
1 select name, employeeID from employee;
```

Решение:

```
1 my $employee_name_and_id_rs =
2   $employee_rs->search(undef, {columns => [qw/name
      employeeid/]});
3 while (my $name_and_id = $employee_name_and_id_rs->next)
    {
4     say $name_and_id->name . ' | ' . $name_and_id->
        employeeid;
5 }
```

Рецепт 9. Создание псевдонимов для имен столбцов и таблиц

Запрос, который мы хотим получить:

```
1 select name as employeeName from employee;
```

Решение:

```
1 my $employee_name_rs =
2   $employee_rs->search(undef, {select => 'name', as => '
      employeeName'});
3 while (my $name = $employee_name_rs->next) {
4     say $name->get_column('employeeName');
5 }
```

Следует обратить внимание, что когда создается алиас, отличный от имени столбца таблицы, то не создается метод-акссесор для этого столбца. Для этого необходимо вызывать метод get_column с именем алиаса.

Если алиас имеет такое же имя, что и значение в select (т.е. есть для него есть метод-аксессор), то можно использовать этот аксессор для столбца.

```
1 my $employee_name_count_max_rs = $employee_rs->search(
2     undef,
3     {
4         select => [
5             'name',
6             {count => 'employeeid'},
7             {max   => {char_length => 'name'}, -as => '
                 longest_name'}
8         ],
9         as => [
10            qw/
11              name
12              employee_count
13              max_name_length
14              /
15        ],
16        group_by => ['name'],
17    }
18 );
19
20 say "name\t\t| count | max name length";
21 while (my $name_count_max = $employee_name_count_max_rs->
     next) {
22     say $name_count_max->name . "\t| "
23       . $name_count_max->get_column('employee_count') . "
          \t| "
24       . $name_count_max->get_column('max_name_length');
25 }
```

Выбор строк с помощью WHERE, DISTINCT, GROUP BY, HAVING

Рецепт 10. Выбор строк с помощью where

Запрос, который мы хотим получить:

```
1 select employeeID, name from employee where job = '
     Программист';
```

Решение:

```
1 my $programmer_rs = $employee_rs->search({job => '
    Программист'});
2 while (my $programmer = $programmer_rs->next) {
3     say $programmer->employeeid . ' | ' . $programmer->
        name;
4 }
```

Рецепт 11. Удаление повторений с помощью distinct

Запрос, который мы хотим получить:

```
1 select distinct job from employee;
```

Решение:

```
1 my $job_rs = $employee_rs->search(undef, {columns => 'job
    ', distinct => 1});
2 while (my $job = $job_rs->next) {
3     say $job->job;
4 }
```

На самом деле, DBIC на выходе генерирует выражение group by, т.е. вместо вышеприведенного запроса будет сгенерирован следующий:

```
1 SELECT me.job FROM employee me GROUP BY me.job:
```

Для подсчета количества значений столбца job без учета повторений, можно реализовать такой запрос:

```
1 select count(distinct job) from employee;
```

Реализация с помощью DBIC:

```
1 my $count = $job_rs->count;
2 say "count: $count";
```

В данном случае DBIC генерирует следующий запрос:

```
1 SELECT COUNT( * ) FROM (SELECT me.job FROM employee me
    GROUP BY me.job) me;
```

Рецепт 12. Выбор групп с помощью having

Запрос, который мы хотим получить:

```
1 select count(*), job from employee group by job having
    count(*) = 1;
```

Решение:

```
1 my $employee_having_rs = $employee_rs->search(
2     undef,
3     {
4         select => [{count => '*', -as => 'count_employee'
             }, 'job'],
5         as     => [
6             qw/
7               count_employee
8               job
9               /
10        ],
11        group_by => ['job'],
12        having   => {count_employee => 1}
13    }
14 );
15
16 say 'count_employee | job';
17 while (my $employee = $employee_having_rs->next) {
18     say $employee->get_column('count_employee') . ' | ' .
          $employee->job;
19 }
```

Выбор строк по первичному ключу

Рецепт 13. Поиск по первичному ключу (PK) с помощью метода find()

Метод `find` производит поиск по первичным ключам, если их значения указаны в качестве списка значений данного PK в порядке, заданном в Result-классе таблицы. Также метод `find` может искать по уникальным ключам, заданным с помощью ссылки на хеш и установленном значении `key` в качестве второго аргумента метода `find`.

Т.е. по сути `find` ищет либо по PK-ключам, указанном в виде списка значений этих ключей в том порядке, в котором они объявлены в Result-классе таблицы, либо по UK-ключам, при этом имя ключа указывается во втором аргументе метода `find`, а сами уникальные поля указываются в первом аргументе, например:

```
$rs->find({employeeid => 7513, clientid => 3}, {key => '
    fk_employee'});
```

Если ключ не указывается, и используется поиск только по заданным столбцам без указания ключа или без использования списка PK-ключей, то результат непредсказуем и в следующих версиях может быть устаревшим. Что имеется в виду под непредсказуемостью: вы пишете вот такой код (`PK(login), pass`):

```
$rs->find({login => 'mylogin', pass => 'mypass'});
            ^                   ^
            |                   |
            PK              Обычное поле
```

на самом деле здесь будет поиск только по PK-ключу login, что может ввести в заблуждение.

Лучше здесь написать так:

```
# Получает список PK-ключей в порядке их определения
# в схеме БД, в данном случае у нас один элемент
$rs->find('mylogin');
```

С другой стороны, можно написать так (login, pass):

```
$rs->find({login => 'mylogin', pass=>'mypass'});
            ^                   ^
            |                   |
       Обычное поле         Обычное поле
```

здесь мы имеем 2 обычных поля (не PK и не UK), которые попадут в условие поиска. Результат тут непредсказуем — либо вернется одно значение и все ок, либо будет выдано предупреждение:

```
"DBIx::Class::Storage::DBI::select_single(): Query
    returned more thanone row.
SQL that returns multiple rows is DEPRECATED for ->find
    and ->single at",
```

которое сообщает, что возврат нескольких строк методами `find` и `single` является устаревшим и не рекомендуется к использованию. С другой стороны, если PK составной и в условии поиска `find` указано только одно поле, то такое предупреждение тоже может выдаться.

Рассмотрим поиск записи по первичному ключу (список PK из одного значения).

Запрос, который мы хотим получить:

```
1 select * from employee where employeeid = 6651;
```

Решение:

```
1 my $employee_find_pk = $employee_rs->find(6651);
2 say 'Employee name: ' . $employee_find_pk->name;
```

Рецепт 14. Поиск в таблице, содержащей составной PK-ключ.

Для PK ключей нужно указывать список значений ключей в том порядке, в котором они указаны в Result-классе заданной таблицы.

Запрос, который мы хотим получить:

```
1 select * from assignment where employeeid = 7513 and
    workdate = '2014-10-25'
```

Решение:

```
1 my $employee_find_by_many_pk = $assignment_rs->find(7513,
     '2014-10-25');
2 say $employee_find_by_many_pk->hours;
```

Рецепт 15. Поиск по уникальному ключу (UK)

Для того, чтобы произвести поиск по уникальному ключу, необходимо указать имя UK-ключа в параметре key метода `find`:

```
1 find({}, {key => 'unique_key'})
```

Более подробно о том, как работает метод find, смотрите в рецепте 13.

Пример запроса:

```
1 select * from assignment where employeeid = 7513 and
    workdate = '2014-10-25';
```

Решение:

```
1 my $assignment_find_clients =
2   $assignment_rs->find({employeeid => 7513, workdate => '
      2014-10-25'},
3     {key => 'fk_employee'});
4 say 'assignment find hours: ' . $assignment_find_clients
    ->hours;
5 say 'assignment find client name: '
6   . $assignment_find_clients->client->name;
7 say 'assignment find employee job: '
8   . $assignment_find_clients->employee->job;
```

Обратите внимание, что метод find позволяет искать в связанных таблицах.

Использование custom-методов для Result- и ResultSet-классов

Рецепт 16. Вызов custom-метода из Result-класса

Вызываем custom-метод name_and_job из Result-класса Company::Schema::Result::Employee.

Custom-метод для Result-класса применяется для каждой строки запроса. Работает по аналогии с методами из класса DBIx::Class::Row.

Решение:

Для одной строки запроса (с помощью метода single):

```
1 my $employee_custom = $employee_rs->single({employeeid =>
    6651});
2 say $employee_custom->name_and_job;
```

Для каждой строки результирующей выборки (с помощью метода `next`):

```
1 while (my $empl_custom = $employee_rs->next) {
2     say $empl_custom->name_and_job;
3 }
```

Рецепт 17. Вызов custom-метода из ResultSet-класса

Вызываем custom-метод `department_client_employee` из ResultSet-класса `Company::Schema::ResulSet::Client`.

Custom-метод для ResultSet-класса применяется для результирующего набора и возвращает нужный результирующий набор (ResultSet). В данном случае мы получили с помощью метода нужные связи + с помощью `search` добавили ограничение на поиск клиента с именем Telco Inc.

Решение:

```
1 my $department_client =
2   $client_rs->department_client_employee->search({name =>
      'Telco Inc'});
3 while (my $dep = $department_client->next) {
4     say 'Address: ' . $dep->address;
5 }
```

Использовние отношений между таблицами (relationships)

Рецепт 18. Использование отношение один-к-одному (**has_one**).

Задача:

```
1 Определить, какие клиенты относятся к служащему 7513.
```

Запрос:

```
1 select c.*
2 from client c
3   join assignment a on a.clientid = c.clientid
4 where a.employeeid = 7513;
```

Решение:

```
1 my $client_for_employee =
2   $client_rs->search({employeeid => 7513}, {join => '
      assignment'});
3 while (my $client = $client_for_employee->next) {
4     say $client->name;
5 }
```

Рецепт 19. Определить, какой служащий имеет клиента 2 (The Bank)

Запрос:

```
1 select a.*
2 from assignment a
3   join client c on c.clientid = a.clientid
4 where a.clientid = 2;
```

Решение:

```
1 my $employee_have_client =
2   $assignment_rs->search({'me.clientid' => 2}, {join => '
      client'});
3 my @employee_ids = $employee_have_client->first->id;
4 say "employeeid: $employee_ids[0]; workdate:
    $employee_ids[1]";
```

Обратите внимание, что аксессор id возвращает массив PK-ключей в порядке их определения в схеме.

Также следует обратить внимание на то, что если вы указываете в условии поиска поле, которое имеется в обеих таблицах, т.е. внешний ключ, то необходимо явно указывать, к какой таблице данное поле принадлежит, например:

```
1 # можно здесь указать me.clientid
```

```
2 my $employee_have_client =
3   $assignment_rs->search({'client.clientid' => 2}, {join
      => 'client'});
```

а вот так уже не будет работать

```
1 # к какой таблице относится clientid?
2 my $employee_have_client =
3   $assignment_rs->search({clientid => 2}, {join => '
      client'});
```

и выдаст ошибку

```
1 DBI Exception: DBD::mysql::st execute failed: Column '
    clientid' in where clause is ambiguous
```

т.е. это ошибка от MySQL, которая не поймет, к какой таблице относится данный clientid.

Рецепт 20. Получить имена служащих, еще не получавших внешних заданий, т.е. служащих, коды которых (employeeid) отсутствуют в таблице assignment

Запрос:

```
1 select e.*
2 from employee e
3   left join assignment a on a.employeeid = e.employeeid
4 where clientid is null;
```

Решение:

```
1 my $employee_assignments =
2   $employee_rs->search({clientid => undef}, {join => '
      assignments'});
3 while (my $employee = $employee_assignments->next) {
4     say $employee->id . ' | ' . $employee->name;
5 }
```

Рецепт 21. Пример отношений много-ко-многим (`many_to_many`)

Задача:

```
Получить все навыки для служащего 7513 (Нора Эдвардс).
```

Запрос:

```
# Сначала получаем служащего с id = 7513
select * from employee where employeeid = 7513;

# Затем если служащий найден, то получаем для него навыки
select * from employeeskills where employeeid = 7513;
```

Решение:

```
my $employee_7513_skills_rs = $employee_rs->find(7513)->
    employee_skills;
while (my $employee = $employee_7513_skills_rs->next) {
    say $employee->skill;
}
```

Подзапросы

Рецепт 22. Найти самое длинное имя служащего (подзапрос, возвращающий одно значение, + функция слева от оператора сравнения)

Запрос, который мы хотим получить:

```
select name from employee where char_length(name) = (
    select max(char_length(name)) from employee);
```

Если не знать, как правильно использовать DBIC, то можно написать такой некрасивый код:

```
my $employee_longest_name = $employee_rs->search(
    {
        'char_length(name)' => {
            '=' => $employee_rs->search(
                undef,
```

```
6                 {
7                     select => [
8                         {
9                             max => {char_length => 'name'
                                 },
10                            -as => 'longest_name'
11                        }
12                    ]
13                }
14            )->get_column('longest_name')->as_query
15        }
16    }
17 );
```

Но разобравшись больше с DBIC, можно найти более элегантное решение:

```
1 my $employee_longest_name = $employee_rs->search(
2     {
3         'CHAR_LENGTH(name)' => {
4             '=' => $employee_rs->get_column('name')->
                 func_rs('char_length')
5               ->get_column('name')->max_rs->as_query,
6         }
7     }
8 );
9 say '--- longest name: ' . $employee_longest_name->single
    ->name;
```

Здесь используются методы:

- `func_rs`, который принимает имя SQL-функции, и добавляет ее в select-запрос для выбранного столбца;
- `max_rs`, который является алиасом для `func_rs('max')`.

Оба метода возвращают ResultSet-объект.

Рецепт 23. Определить, кто из программистов работал над выполнением внешних заданий (from (select...))

Запрос, который мы хотим получить:

```
select programmer.name
from (select employeeID, name from employee where job='Программист') as programmer, assignment
where programmer.employeeID = assignment.employeeID;
```

Чтобы все это дело заработало, нужно переделать запрос на вот такой:

```
SELECT employee.name
FROM (
  SELECT employee.name
  FROM assignment me
    JOIN employee employee ON employee.employeeid = me.employeeid
    WHERE ( job =  'Программист' )
) employee;
```

Решение:

```
my $assignment_prog_rs = $assignment_rs->search_related(
    'employee',
    {job     => 'Программист'},
    {columns => ['name']}
)->as_subselect_rs->search(undef, {columns => ['name']});

while (my $programmer = $assignment_prog_rs->next) {
    say 'programmer name: ' . $programmer->name;
}
```

Т.е. суть в том, что DBIC позволяет создать только такие встроенные представления (под встроенными представлениями понимается вложенный select в оператор from, т.к. по сути он (select) возвращает набор значений), в которых будут только эти самые встроенные представления. В этот же from не получится ничего добавить стандартными документированными способами. По крайней мере документации по атрибуту from я не нашел, хотя он (атрубут) используется внутри самого DBIC.

Хотя как по мне, так проку от такого запроса нет, т.к. можно просто опуситить внешний select, оставить только запрос внутри from и все будет работать так же. Если кто знает как реализовать на DBIC запрос вида:

```
select programmer.name
from (
```

```
3           select employeeID, name
4           from employee
5           where job='Программист'
6        ) as programmer,
7        assignment
8 where programmer.employeeID = assignment.employeeID;
```

без преобразования для использования с join-ом и с двумя таблицами, просьба указать в комментариях.

Рецепт 24. Определить, кто из служащих потратил больше всех времени на выполнение задания (join + подазпрос, возвращающий одно значение)

Запрос, который мы хотим получить:

```
1 select e.employeeID, e.name
2 from employee e, assignment a
3 where e.employeeID = a.employeeID
4 and a.hours = (select max(hours) from assignment);
```

Данный запрос лучше переделать на следующий:

```
1 select e.employeeID, e.name
2 from employee e
3   join assignment a on e.employeeID = a.employeeID
4 where a.hours = (select max(hours) from assignment);
```

Решение:

```
1 my $employee_max_busy_rs = $employee_rs->search(
2     {
3         hours => {
4             '=' => $assignment_rs->get_column('hours')->
                  max_rs->as_query,
5         }
6     },
7     {join => 'assignments'}
8 );
9
10 while (my $employee = $employee_max_busy_rs->next) {
11     say $employee->id . ' | ' . $employee->name;
12 }
```

Рецепт 25. Определить служащих, которые не имели внешних заданий (NOT IN)

Запрос, который мы хотим получить:

```
1 select name from employee
2 where employeeID not in (select employeeID from
    assignment);
```

Решение:

```
1 my $employee_not_have_assignment_rs = $employee_rs->
    search(
2     {
3         employeeid => {
4             -not_in => $assignment_rs->get_column('
                employeeid')->as_query,
5         }
6     }
7 );
8
9 while (my $employee = $employee_not_have_assignment_rs->
    next) {
10     say $employee->name;
11 }
```

Рецепт 26. Определить служащих, которые не входят в заданное множество (NOT IN)

Запрос, который мы хотим получить:

```
1 select name from employee where employeeID not in (6651,
    1234);
```

Решение:

```
1 my $employee_not_in_set = $employee_rs->search(
2     {
3         employeeid => {
4             -not_in => [qw/6651 1234/],
5         }
6     }
7 );
8
```

```
9 while (my $employee = $employee_not_in_set->next) {
10     say $employee->name;
11 }
```

Рецепт 27. Получить список служащих, которые никогда не работали над внешними заданиями (NOT EXISTS)

Запрос, который мы хотим получить:

```
1 select e.name, e.employeeID
2 from employee e
3 where not exists (select * from assignment where
    employeeID = e.employeeID);
```

Решение:

```
1 my $employee_not_work_with_assignments = $employee_rs->
    search(
2     {
3         -not_exists => $assignment_rs->search(
4             {
5                 'a.employeeid' => {-ident => 'e.
                    employeeid'}
6             },
7             {
8                 alias   => 'a',
9                 columns => [qw/e.employeeid e.name/]
10            }
11        )->as_query,
12    },
13    {
14        alias   => 'e',
15        columns => [qw/e.employeeid e.name/],
16    }
17 );
18
19 while (my $employee = $employee_not_work_with_assignments
    ->next) {
20     say $employee->name;
21 }
```

Вот что генерит DBIC:

```
1 SELECT e.employeeid, e.name
2 FROM employee e
```

```
3 WHERE ( (NOT EXISTS (SELECT e.employeeid, e.name FROM
    assignment a WHERE ( a.employeeid = e.employeeid ))) ):
```

Следует отметить, что нужно аккуратно использовать алиасы, т.к. по умолчанию все привыкли работать с `me`, поэтому для основного запроса можно не делать `alias => 'e'`, а везде писать `me.column_name`. Здесь алиасы приведены только для примера.

Как можно видеть, DBIC позволяет писать как join-запросы, так и подзапросы. На мой взгляд, сила DBIC — в связях с помощью join, используя которые, можно не писать ключи в условиях where с помощью `-ident` (первый аргумент `search()`, например `{id1 => {-ident => me.id2}}`), а просто написать `join => 'joining_table_name'`.

Ограничение результатов поиска с помощью LIMIT

Рецепт 28. Ограничение LIMIT n;

Запрос, который мы хотим получить:

```
1 select * from employeeSkills limit 5;
```

Решение:

```
1 my $skills_limit = $employeeskill_rs->search(undef, {rows
    => 5});
2 while (my $employee = $skills_limit->next) {
3     say $employee->skill;
4 }
```

Рецепт 29. Ограничение LIMIT n, m;

Запрос, который мы хотим получить:

```
1 select * from employeeSkills limit 5, 3;
```

Решение:

```
1 # здесь получается ($first, $last) включительно,
2 # т.е. у нас получается 3 записи: 5, 6, 7
3 my $skills_limit_with_slice = $employeeskill_rs->slice(5,
     7);
4 while (my $employee = $skills_limit_with_slice->next) {
5     say $employee->skill;
6 }
```

Рецепт 30. Разделение на страницы (pagination).

Формула расчета:

```
1 limit ($page-1)*$rows, $rows
```

Запрос:

```
1 select * from employeeSkills limit 4, 2;
```

Решение:

```
1 my $skills_limit_with_page =
2   $employeeskill_rs->search(undef, {page => 3, rows =>
      2});
3 while (my $employee = $skills_limit_with_page->next) {
4     say $employee->skill;
5 }
```

CRUD для строк

Здесь будут рассмотрены CRUD-операции над Result-объектами, которые возвращаются ResultSet-методами: `create`, `find`, `next` и `all`, т.е. по сути операции над отдельными строками.

Рецепт 31. `insert`

Запрос, который мы хотим получить:

```
1 insert into employee (departmentid, employeeid, job, name
    ) values ('145', '9739', 'Главный программист', 'Билл
    Гейтс');
```

Решение:

```
1 my $new_row_for_employee = $employee_rs->new(
2     {
3         employeeid   => 9739,
4         name         => 'Билл Гейтс',
5         job          => 'Главный программист',
6         departmentid => 145
7     }
8 );
9
10 my $insert_new_row_to_employee = $new_row_for_employee->
   insert;
```

Рецепт 32. **delete**

Запрос:

```
1 # Ищем служащего с id = 9789
2 select * from employee where employeeid = 9739;
3
4 # Если нашли, то удаляем
5 delete from employee where employeeid = 9739;
```

Решение:

```
1 my $delete_employee_row = $employee_rs->find(9739)->
   delete;
```

Рецепт 33. **update**

Запрос:

```
1 begin work
2 update client set name = 'Telco Inc' where clientid = 1;
3 commit
```

Решение:

```
1 my $update_client_row = $client_rs->find(1)->update({name
    => 'Telco Inc'});
```

Рецепт 34. **`update_or_insert`** | **`insert_or_update`**

Обновляет объект, если он есть в базе (на основе `in_storage`-метода), иначе заносит его в базу.

Запросы:

```
--- update_or_insert: update
select * from department where departmentid = 130;
update department
set name = 'Отдел управления и маркетинга'
where departmentid = 130;

--- update_or_insert: update
select * from department where me.departmentid = 130;
update department
set name = 'Отдел маркетинга'
where departmentid = 130;

--- update_or_insert: insert
select * from department where me.departmentid = 155;
insert into department (departmentid, name)
values ('155', 'Отдел информационных технологий');
```

Решение:

```
say '--- update_or_insert: update';
my $new_row_for_department =
  $department_rs->find_or_new({departmentid => 130});
$new_row_for_department->name('Отдел управления и
   маркетинга');
$new_row_for_department->update_or_insert;    # update

say '--- update_or_insert: update';
$new_row_for_department =
  $department_rs->find_or_new({departmentid => 130});
$new_row_for_department->name('Отдел маркетинга');
$new_row_for_department->update_or_insert;    # update

say '--- update_or_insert: insert';
$new_row_for_department =
  $department_rs->find_or_new({departmentid => 155});
$new_row_for_department->name('Отдел информационных
   технологий');
$new_row_for_department->update_or_insert;    # insert
```

Метод `insert_or_update` является алиасом для `update_or_insert`.

CRUD

Рецепт 35. **create**

Запросы:

```
insert into client ( address, contactnumber,
   contactperson, name)
values ('ул. Кандаурова, 25/3', '123456789777', 'Наталья
   Ветлицкая', 'Азовский рынок');
```

Решение:

```
my $create_client = $client_rs->create(
    {
        name          => 'Горбушка',
        address       => 'ул. Барклая, 8',
        contactperson => 'Наталья Ветлицкая',
        contactnumber => '123456789777'
    }
);
```

Рецепт 36. **delete**

Запрос:

```
delete from client where name = 'Горбушка';
```

Решение:

```
my $delete_client =
  $client_rs->search({name => 'Горбушка'})->delete;
```

Рецепт 37. **populate**

Метод вставляет набор записей в рамках одной транзакции.

Так будет работать в непустом (non-void) (скалярном или списковом) контексте:

```
1 BEGIN WORK
2 INSERT INTO department ( departmentid, name) VALUES ('132
    ', 'Инженерный отдел');
3 INSERT INTO department ( departmentid, name) VALUES ('134
    ', 'Отдел легкой промышленности');
4 INSERT INTO department ( departmentid, name) VALUES ('135
    ', 'Отдел тяжелой промышленности');
5 COMMIT
```

т.е. это обычная обертка над методом `create` с добавлением транзакций.

Решение:

```
1 my $populate_client = $department_rs->populate(
2     [
3         [qw/departmentid name/],
4         [132, 'Инженерный отдел'],
5         [134, 'Отдел легкой промышленности'],
6         [135, 'Отдел тяжелой промышленности'],
7     ]
8 );
```

А так будет работать в пустом (void) контексте:

```
1 BEGIN WORK
2 INSERT INTO department ( departmentid, name) VALUES ( ?,
    ? ): '__BULK_INSERT__'
3 COMMIT
```

Здесь используется метод `execute_for_fetch` из DBI, что позволяет ускорить вставку данных за счет непосредственной заливки данных в БД. Кроме того, есть и недостаток этого подхода — если используются генерируемые с помощью DBIC столбцы, то данные для этих столбцов не будут генерироваться. Если это необходимо, то лучше использовать вызов метода в скалярном или в списковом контекстах.

Решение:

```
1 $department_rs->populate(
2     [
3         [qw/departmentid name/],
```

```
4         [137, 'Маркетинговый отдел'],
5         [139, 'Отдел электроники'],
6         [141, 'Отдел бытовой техники'],
7     ]
8 );
```

Рецепт 38. **delete_all**

Вызывает поиск нужных значений и удаление в рамках одной транзакции:

```
1 BEGIN WORK
2 SELECT me.departmentid, me.name FROM department me WHERE
    departmentid IN ( '132', '134', '135');
3 DELETE FROM department WHERE departmentid = '132';
4 DELETE FROM department WHERE departmentid = '134';
5 DELETE FROM department WHERE departmentid = '135';
6 COMMIT
```

Решение:

```
1 my $delete_all_department = $department_rs->search(
2     {
3         departmentid => {-in => [qw/132 134 135 137 139
             141 150 155/]}
4     }
5 )->delete_all;
```

Рецепт 39. **update**

Обновляет данные для всего ResultSet-а

Решение:

```
1 my $update_employee =
2   $employee_rs->search({departmentid => 128})->update({
      vacation => 'Yes'});
```

DBIC выдает следующий код:

```
1 UPDATE employee SET vacation = 'Yes' WHERE ( departmentid
      = '128');
```

CRUD с поиском

Рецепт 40. `find_or_new`

Ищет существующую запись из результирующего набора с помощью find. Если запись не существует, создается и возвращается новый объект. Объект не будет сохранен в БД, если не будет вызван метод `insert` из `DBIx::Class::Row`. Пример использования см. в рецепте `update_or_insert`.

Рецепт 41. `find_or_create`

Пытается найти запись по primary key или unique. Если запись не существует, данные заносятся в БД. Следует обратить внимание на то, что можно и нужно в качестве аргумента указать все поля, нужные для занесения, а не только PK или UK. Т.е. `find` работает как обычно — в получаемом хеше обрабатывает только PK либо UK (если во втором аргументе указан key), если запись не найдена, то заданный в качестве аргумента хеш записывается в БД. Также следует учесть, что в данном случае не используются транзакции, поэтому нужно самим позаботится о создании транзации во избежание ситуации гонки, т.е. когда данные в БД обновились после `find`, но перед `create`.

Решение:

```
1 my $department_create = $department_rs->find_or_create(
2     {
3         departmentid => 153,
4         name =>
5           'Отдел информационных технологий'
6     }
7 );
```

Рецепт 42. `update_or_new`

Решение:

```
1 my $department_it = $department_rs->update_or_new(
2     {
3         departmentid => 150,
4         name         => 'Отдел IT'
5     }
6 );
7
8 # если запись не найдена, то заносим ее в БД
9 $department_it->insert unless $department_it->in_storage;
```

По аналогии с `find_or_new`, но если строка найдена, то она немедленно обновляется через `$found_row->update(\%col_data)`.

Рецепт 43. update_or_create

Решение:

```
1 my $department_update_it = $department_rs->
    update_or_create(
2     {
3         departmentid => 153,
4         name => 'Отдел программирования станков с ЧПУ'
5     }
6 );
```

По аналогии с `find_or_create`, но если строка найдена, то она немедленно обновляется с помощью `$found_row->update(\%col_data)`.

Заключение

В заключение хочется обратить внимание на достоинства и недостатки `DBIx::Class`.

Достоинства:

- активная разработка;
- расширяемость;

- большое количество расширений, хелперов.

Недостатки:

- нет возможности использовать `DISTINCT`, т.к. DBIC генерирует `GROUP BY`;
- не документирована возможность создания сложных подзапросов с несколькими вложенными select в оператор from с добавлением других таблиц.

■ *Вячеслав Коваль*

6. Обзор CPAN за ноябрь 2014 г.

Рубрика с обзором интересных новинок CPAN за прошедший месяц.

Статистика

- Новых дистрибутивов — 222
- Новых выпусков — 782

Новые модули

- Redis::Jet

Ещё одна XS-реализация клиента Redis. Модуль находится в стадии активной разработки, но судя по представленным бенчмаркам имеет превосходство над всеми существующими реализациями (`Redis`, `Redist::Fast`, `Redis::hiredis`).

- Rest::Client::Builder

Обилие веб-сервисов, предоставляющих REST API, требует рутинного создания типовых REST-клиентов для этих API. Модуль `Rest::Client::Builder` предоставляет магический инструмент для простого создания подобных REST-клиентов в полном соответствии с ООП-парадигмой:

```
1  package My::Magic::API;
2  use base qw(Rest::Client::Builder);
3
4  sub new {
5      bless $_[0]->SUPER::new( {
6          on_request => sub { "@_" }
7      }, 'http://example.com/api' ), $_[0]
8  }
9
```

```
10  my $api = My::Magic::API->new();
11
12  print $api->foo(1)->bar->get('baz=qux');
13  # GET http://example.com/api/foo/1/bar baz=qux
14
15  print $api->blah->post('postData');
16  # POST http://example.com/api/blah postData
```

- Test::TempDir::Tiny

Нередко в тестах требуется создавать временные каталоги. Модуль `Test::TempDir::Tiny` предоставляет функцию `tempdir()` удобную для использованию именно в процессе тестирования. Каждый тест (файл *.t) получает свой временный корневой каталог. Если тест падает, то временный каталог не удаляется, предоставляя возможность просмотреть файлы в этом каталоге. При последующем запуске теста существующий временный каталог теста предварительно удаляется. Если все тесты успешно пройдены, корневой временный каталог удаляется. Модуль `Test::TempDir::Tiny` имеет зависимости только от core-модулей (что и означает суффикс Tiny).

- recommended

С помощью модуля `recommended` можно загружать другие модули, если они доступны на момент запуска. В целом это аналог `use`, но без фатальной ошибки в случае отсутствия модуля.

```
1  use recommended 'Foo::Bar', {
2    'Bar::Baz' => '1.23',
3    'Wibble'   => '0.14',
4  };
5
6  # Действия, если Foo::Bar был загружен
7  if ( recommended->has('Foo::Bar') ) {
8      ...
9  }
```

- Qstruct

`Qstruct` — это ещё один бинарный формат сериализации данных. В отличие от Storable/Sereal/CBOR этот формат требует схему, что делает его похожим на ASN.1 или ProtocolBuffers. Многие идеи Qstruct были позаимствованы из формата Cap'nProto. Основная идея — приблизить скорость работы к скорости работы struct в языке C, что достигается за счёт того, что формат хранения данных в памяти совпадает с форматом, хранимым на диске или передаваемым по сети. При этом формат является максимально переносимым (выбрано little-endian представление на всех платформах).

- Plack::Debugger

`Plack::Debugger` — это удобный отладчик для ваших PSGI-приложений. Отладчик подключается как middleware, после чего в выводимый html-контент перед закрывающим тегом `</body>` добавляется код панели отладчика, в которой можно посмотреть временны́е характеристики генерации страницы, потребляемую память, предупреждения. Также можно отслеживать все ajax-запросы, выполняемые приложением. Отладчик сохраняет всю собранную информацию о запросах в виде json-файлов для последующего анализа.

- Database::Sophia

Модуль `Database::Sophia` представляет собой байндинг к базе данных Sophia. Sophia — это встраиваемая база данных для хранения ключей-значений. Судя по представленным бенчмаркам, Sophia обходит в производительности схожую по принципам работы LevelDB.

- Gazelle

Очередной высокопроизводительный Plack-сервер с красивым названием `Gazelle`. Представлены сравнительные бенчмарки, сравнивающие количество обрабатываемых запросов в секунду у Gazelle со

Starman и Starlet. Gazelle демонстрирует почти в два раза большую производительность чем конкуренты, а на тесте с выдачей hello world приближается к скорости отдачи статики в nginx.

- App::Implode

Утилита implode позволяет запаковать Perl-приложение со всеми его зависимостями в один исполняемый Perl-скрипт. Принцип работы достаточно прост: при помощи carton на основе cpanfile загружаются все зависимости приложения во временный каталог, затем этот каталог архивируется (tar) и упаковывается (bzip2). Полученный архив вносится внутрь скрипта после секции __END__, а в начале скрипта добавляется секция BEGIN, которая отвечает за распаковку архива во временный каталог и добавления путей в PATH и PERL5LIB.

- Regexp::Lexer

Regexp::Lexer — это завершающий компонент для Perl::Lint , призванный заменить Regexp::Parser, предназначенный для лексического разбора регулярных выражений.

- Panda::URI

Panda::URI — это альтернатива модулю URI, написанная на языке C. Утверждается, что в некоторых операциях она превосходит в скорости URI на два порядка.

Обновлённые модули

- Devel::StackTrace 2.00

Вышел новый мажорный релиз модуля `Devel::StackTrace` — объектного представления трассировки вызовов. В новом релизе присутствует несколько несовместимых изменений (удалены некоторые устаревшие методы).

- Pithub 0.01028

В ноябре был обновлён модуль `Pithub` для доступа к API Github. В новом релизе много исправлений ошибок и несколько полезных оптимизаций. В частности, за счёт использования условных запросов и кеширования снижается сетевая нагрузка и ускоряется обработка запросов.

- DBI 1.632

Новый релиз `DBI` содержит несколько исправлений, включая исправление уязвимости в модуле `DBD::File`. `DBD::File`, несмотря на атрибут `f_dir`, первоначально искал файл относительно текущего каталога. Это также затрагивает все модули, использующие `DBD::File`, например, `DBD::CSV`. Также в документацию `DBD::Proxy` и `DBI::ProxyServer` добавлено предупреждение о том, что их использование небезопасно, поскольку для сериализации данных используется модуль `Storable`, который небезопасно использовать с недоверенными источниками.

- Minion 1.02

Вышел первый мажорный релиз модуля `Minion` для управления очередями задач. На данный момент он уже не является экспериментальным и поддерживает два бекенда: файлы и PostgreSQL.

- YAML-LibYAML 0.54

Вышло обновление XS-реализации парсера языка разметки YAML. Данный релиз исправляет ошибку в безопасности библиотеки libyaml. При разборе специального сформированного YAML-файла может произойти вызов `assert()`, что приведёт к аварийному завершению программы по сигналу `SIGABRT`. Например:

```
$ perl —MYAML::XS —e 'eval { Load qq! x: "\n"x! };'

perl: scanner.c:1113: yaml_parser_save_simple_key:
   Assertion `parser—>simple_key_allowed || !required'
   failed.
```

Как видно, программа завершится аварийно, что может быть использовано для DoS-атак против систем, использующих libayml при разборе данных в YAML-формате.

■ *Владимир Леттиев*

7. Интервью с Олафом Алдерсом (Olaf Alders)

Олаф Алдерс (Olaf Alders) — канадский Perl-программист, создатель MetaCPAN

Когда и как научился программировать?

Уже не помню по какой-то причине, но, когда я был в начальных классах, я посетил компьютерные курсы для начинающих в библиотеке. Это был мой первый опыт программирования.

По-настоящему же программировать я научился в старших классах. Наш класс занимался Waterloo BASIC в сети Commodore 64s. Дома у нас не было компьютера, поэтому я не применял свои навыки нигде кроме школы. Это было весело, но я никогда не думал, что буду заниматься программирование профессионально.

В университете я начал учиться по научной программе. Мой единственный компьютерный курс был на Fortran 77. Я его ненавидел. В конечном итоге я перешел из науки в гуманитарии. Тогда я снова занялся компьютерами и начал учить Perl.

Какой редактор используешь?

Уже в течение нескольких лет я использую Vim. Я не продвинутый пользователь, но у меня получается. https://github.com/oalders/dot-files/blob/master/vim/vimrc. Я годами с удовольствием пользовался разными GUI-редакторами, но, после того как один программист на прошлой работе показал мне несколько фишек vim, меня зацепило.

Когда и как познакомился с Perl?

Познакомился с Perl в университете. Я изучал греческий и латинский и сделал сайт для «Клуба классики». Написал CGI-счетчик посещений. Это было весело, затем я написал сервис, основанный на том счетчике, и разослал e-mail всем своим друзьям. Новость быстро распространилась, и в конце концов у меня были тысячи посеще-

ний. Дошло до того, что мне пришлось брать плату для покрытия расходов на хостинг. Вскоре я начал настоящий бизнес, и все было написано на Perl.

С какими другими языками нравится работать?

Я немного писал на Objective-C, когда работал над iCPAN. Когда нужно было, немного ковырялся с шелл-скриптами и JavaScript. Сейчас начал читать книгу про Go.

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

Не знаю есть у меня твердые аргументы чем Perl лучше или хуже других языков, но для меня он удобен. Мне нравится, как я могу себя выражать.

Также хочу сказать, что мне действительно нравятся люди в Perl. Когда мне нужна была помощь, у меня был только положительный опыт. У меня появились друзья в “сообществе”, и для меня оно комфортно. Конференции YAPC::NA всегда очень интересные. Также у меня была замечательная возможность посетить QA-хакатоны в Париже и Лионе, и я также планирую посетить хакатон в Берлине в 2015 г. Такие мероприятия, когда удается их посещать, остаются в памяти.

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

Я не дизайнер языков, но так как все больше и больше появляется встроенных систем, мне кажется, что возможность встраиваемости будет большим преимуществом.

Как начался MetaCPAN, в чем была твоя роль?

Однажды я решил научиться писать приложения для iPhone. Мне хотелось сделать копию CPAN-документации на своем телефоне, потому что мне нравится читать про разные модули. Я добирался на работу в метро, и там не было интернет-связи, и я никак не мог читать CPAN. Потом убедил друга поработать над этим приложением.

Я сосредоточился на парсинге Pod из minicpan и добавлением его в базу SQLite. Углубляясь все больше и больше в проблему, я понял, что это было не так просто, и что на тот момент решения не существовало. С этой проблемой сталкивались многие и до меня. И так как модули на CPAN постоянно обновляются, мне казалось, что эту проблему можно решить с помощью веб-сервиса.

На встрече перл-монгеров Торонто мы говорили об этой проблеме целый вечер. Один из участников упомянул Elasticsearch, о котором я вообще ничего не слышал. Многие также дали мне по $20. В течение следующих шести недель я проводил вечера за извлечением данных из базы iCPAN и добавлением их в Elasticsearch. Я хостил все на Rackspace и оплачивал собранными средствами. Так как Elasticsearch предоставляет REST-интерфейс из коробки, в конце шестинедельного периода у меня был работающий сервис.

Марк Юбенвиль (Mark Jubenville) (который работал со мной над iCPAN) написал http://search.metacpan.org на чистом JavaScript. Мы использовали его для тестирования API.

В это же время появился Мориц Онкен (Moritz Onken). В рамках проекта Google Summer of Code он написал http://metacpan.org. С тех пор MetaCPAN стал намного популярнее, чем я ожидал. Мы прошли путь от небольшого сервера в облаке до шести серверов в двух дата-центрах. У нас есть ядро разработчиков и много коммитеров. Этот проект требует много времени, у него много составляющих, но это интересно.

Ожидал ли ты, что MetaCPAN практически заменит SCO?

Нет, и вообще он на самом деле не сделал этого в глобальном масштабе. В узких кругах MetaCPAN стал более популярным, но у нас нет, наверное, и 20% того трафика, что есть у http://search.cpan.org. Габор Сабо (Gabor Szabo) может это подтвердить или опровергнуть. Это скорее всего связано с тем, что первые результаты Google-выдачи ведут прямо на search.cpan.org. Есть также множество людей, которые клянутся, что search.cpan.org просто лучше. Габор даже начал работать над SCO-клоном, который будет внутри использовать MetaCPAN: https://github.com/szabgab/MetaCPAN-SCO.

Этой зимой мы будем работать над улучшением поиска, поэтому у меня есть надежды, что скоро сайт станет еще удобнее.

Также хочу упомянуть, что MetaCPAN не планировался как замена SCO. Всегда хорошо иметь выбор.

Что было самой большой проблемой при разработке MetaCPAN?

Самой большой проблемой было найти разработчиков, которые знали или хотели учить Elasticsearch. Это отличная утилита для разработки и внедрения. Она может делать очень сложные и впечатляющие вещи, но в некоторых случаях требует изрядного умения, чтобы себя проявить. Также приходится устанавливать и настраивать множество сервисов, чтобы отобразить наше окружение в продакшене. Впрочем, мы это практически решили. Лео Лэпворт (Leo Lapworth) потратил некоторое время на переезд на Puppet и создание виртуальных машин Vagrant для разработки. Поэтому теперь можно очень быстро запустить собственный MetaCPAN https://github.com/CPAN-API/metacpan-developer.

Как начать помогать работать над MetaCPAN?

Вначале хочу сказать, что всегда можно получить помощь на канале `#metacpan` в сети irc.perl.org. Это очень дружественный канал с небольшим количеством экспертов, поэтому можно себя комфортно чувствовать, задавая вопросы. Мы всегда стараемся разбираться вместе.

Если вы только хотите работать над сайтом, это довольно просто:

```
1 git clone https://github.com/CPAN-API/metacpan-web.git
2
3 cd metacpan-web
4
5 carton install
6 ./bin/prove t
7 carton exec plackup -p 5001 -r
```

Если вы хотите работать над API, тогда лучшим способом будет начать с ВМ https://github.com/CPAN-API/metacpan-developer.

В каком состоянии сейчас находится iCPAN? Можешь объяснить,

что это вообще такое?

iCPAN — универсальное iOS-приложение, это значит, что оно работает на iPhone и iPad, но с некоторыми изменениями в интерфейсе. В нем находится практически весь Pod самых актуальных версий CPAN-модулей. Приложение может быть полезным в самолете, в метро или очереди в банке. Не нужно интернет-соединения. Также оно позволяет добавлять модули в избранное (на iPhone как минимум). Однако, iCPAN уже несколько месяцев нет на App Store. Если не платить взносы Apple, они убирают приложение из магазина, что и произошло со мной. Недавно я думал об оживлении проекта. Если достаточно людей поддержат упомянутый далее тикет, я снова выложу приложение https://github.com/oalders/iCPAN/issues/19.

Где сейчас работаешь? Сколько времени проводишь за написанием Perl-кода?

Я работаю в MaxMind.com. Большинство времени, помимо обсуждений, я провожу за написанием Perl-кода, и у меня есть возможность содействовать открытым проектам во время своей работы. Мне очень повезло в этом смысле.

Стоит ли советовать молодым программистам учить сейчас Perl?

Считаю, что да. Я не думаю, что стоит советовать учить только один язык, но Perl это хороший язык в инструментарии молодого программиста. В течение нескольких лет я был вовлечен в программы Google Summer of Code и GNOME's Outreach Program for Women и я видел, как многие талантливые люди делали интересные вещи на Perl.

Я не уверен, что количество работающих Perl-разработчиков уменьшается, но они явно становятся старше. Мы можем решить это привлечением молодых людей программированию на Perl. У "Girls Who Code", похоже, правильная идея http://girlswhocode.com/.

Моим дочерям сейчас 3 и 5 лет, им еще рано начинать заниматься Perl. Если у них появится интерес к программированию, я бы начал с чего-нибудь наподобие Scratch, но обязательно бы познакомил их

с Perl в свое время.

Вопросы от читателей

Как у тебя получается совмещать работу программистом, делать вклад в открытые проекты и играть в музыкальной группе?! Помогает ли создание музыки программированию?

Я совмещаю все три вещи очень плохо. Я провожу не так много времени, играя музыку, сколько хотелось бы. Вообще, я стараюсь проводить большинство своего свободного времени с моими детьми, но мне повезло, что я могу участвовать в открытых проектах непосредственно во время работы. Все остальное я пытаюсь втиснуть в оставшееся свободное время.

Не знаю, помогает ли музыка с программированием. Я учил древнегреческий и латинский в университете, и это мне очень помогло. Если вы можете читать древнегреческий, вам проще разобраться в чужом коде.

Почему http://wundersolutions.com выдает nginx-страницу?

Упс! Только что починил. :)

Dancer или Mojolicious?

Mojolicious. Недавно на работе у нас было длительное обсуждение фреймворков. Я провел много времени, разбираясь в разных вариантах, и делал даже доклад о том, что выяснил. Наша группа затем решила перейти на Mojolicious в следующих проектах. Также я использую его для своих личных поделок, и мне очень нравится, что так просто начать разработку и запустить сайт.

Кроме безболезненного старта, меня сильно впечатлило качество документации. Мне не приходится напрягаться и искать, что мне нужно, и я это очень ценю.

Начнешь ли снова писать статьи?

Вообще-то я уже начал. У меня в запасе есть несколько вещей, о ко-

торых хочется написать, поэтому множество статей не за горами.

■ *Вячеслав Тихановский*