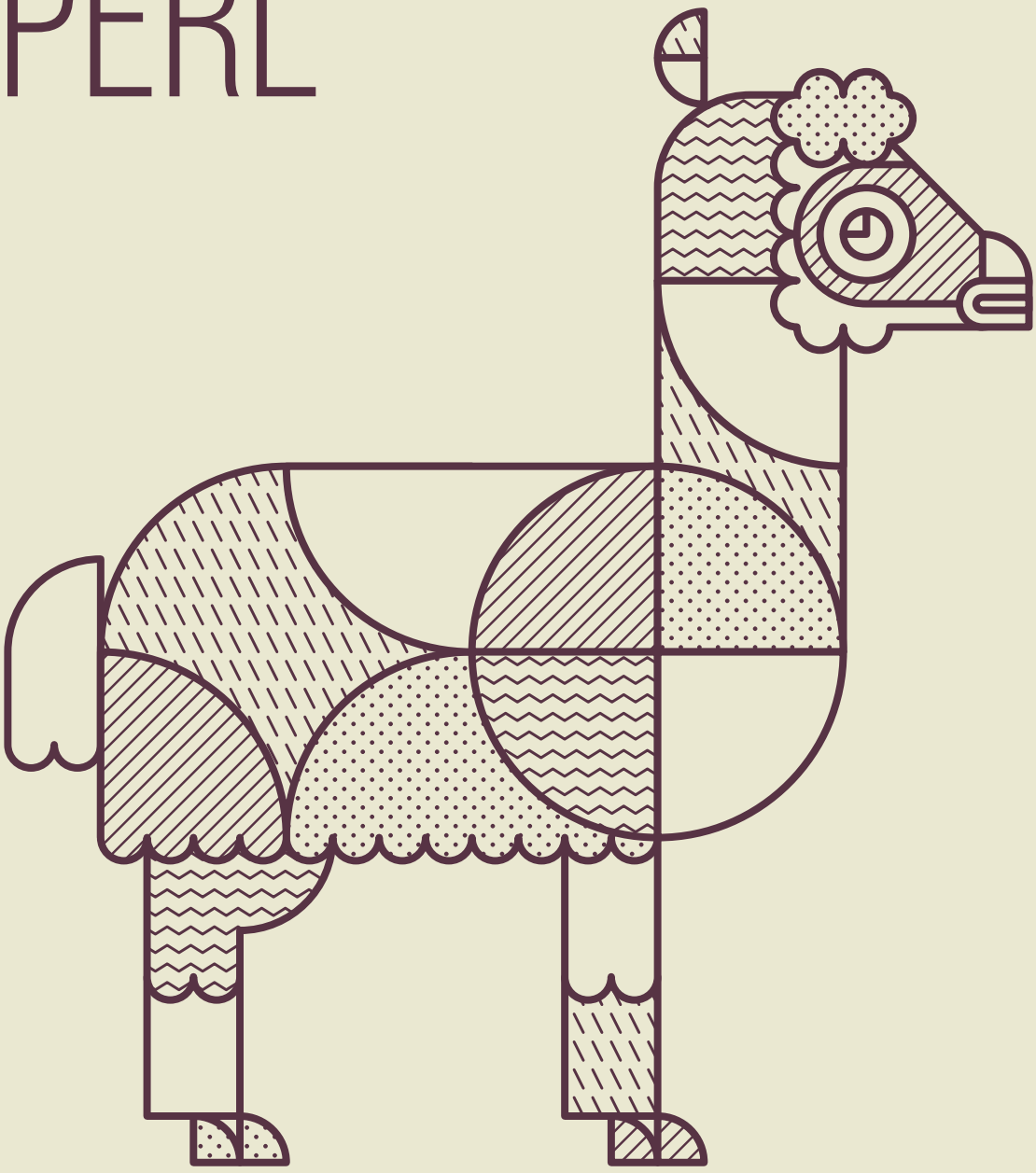
PRAGMATIC PERL

14



04/2014

pragmaticperl.com

Pragmatic Perl 14

pragmaticperl.com

Выпуск 14. Апрель 2014

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

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

Авторы статей: Алексей Мележик, Дмитрий Шаматрин, Владимир Леттиев

Обложка: Марко Иваныk

Корректор: Андрей Шитов

Выпускающий редактор: Вячеслав Тихановский

Ревизия: 2014-11-29 16:19

Оглавление

1 От редактора

Друзья, нам очень важно ваше мнение. Оставляйте комментарии к статьям, задавайте вопросы. Таким образом мы сможем планировать материалы для следующих выпусков исходя из ваших пожеланий.

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

Приятного чтения.

■ *Вячеслав Тихановский*

2 Тестирование в Perl. Лучшие практики

Рассмотрены основные практики для улучшения качества тестирования в Perl

В моей компании мы очень серьезно относимся к тестированию. За годы работы и реализации проектов различной сложности накопилось некоторое количество лучших практик, которые помогают сэкономить на запуске, поддержке и внедрении решений для бизнеса. Для разработчиков это существенно упрощает проведение рефакторинга, поиска багов, мотивирует к написанию более качественного кода. У нас нет отдела тестирования, сам процесс разработки и является тестированием.

В данной статье мы рассмотрим разработку

через тестирование. Разумеется, это лишь часть процесса и никак не исключает других видов тестирования (функциональное тестирование, тестирование безопасности и прочее). Везде, где не указано иное, подразумевается юнит-тестирование (тестирование отдельных классов).

Основные преимущества разработки через тестирование

- Интерфейс создается автоматически.

  Нет необходимости продумывать интерфейс класса. Он сам собой вырисовывается во время использования (тестирования).

- Имплементируется только необходимое.

Нет кода, который не используется. Все, что не протестировано, выкидывается.

- Система разрабатывается небольшими шагами.

  Нет возможности реализовать большой кусок кода, так как всегда требуется написать тест до самого кода.

- Поощряется написание модульного кода.

  Модульные тесты пишутся на классы. Чем несвязаннее классы — тем легче их тестировать.

- Ошибки проектирования выявляются как можно раньше.

  Во время тестирования на ранних этапах проявляются все сложности и

неудобства неправильного проектирования, будь-то сложность работы класса или подсистемы с другими классами и подсистемами, невозможность тестирования или так далее.

- Модуль работает.

  Есть уверенность в том, что модуль хоть как-то работает до его внедрения.

- Возможность рефакторинга.

  При наличии тестовой базы рефакторинг становится безопасным. Изменение поведения и наличие регрессий контролируется тестами.

Основные недостатки разработки через тестирование

- Одни и те же ошибки могут быть оставлены как в коде, так и в тестах.

  Тест может тестировать неправильную логику работы модуля.

- Ложная уверенность в работоспособности системы.

  При наличии большого количества тестов может сложиться ложная уверенность в отсутствии ошибок в системе.

- Больше кода для поддержки

  Тесты это тоже код и его нужно поддерживать. Это время.

- Тесты могут быть хрупкими и ломаться при каждом незначительном изме-

нении кода.

Большинство недостатков можно побороть, следуя лучшим практикам.

Лучшие практики

Следование циклу разработки

Разработка через тестирование должна следовать циклу: красный, зеленый, рефакторинг. Что обычно означает:

- Написать тест и убедиться, что тест не проходит.

  Это необходимо для избежания случаев, когда тест неправильный.

Например, вы нашли ошибку в коде, знаете как ее исправить, написали тест, но не убедились, что именно этот тест выявляет ошибку и исправили код.

- Написать класс и убедиться, что тест проходит.

- Отрефакторить написанный код.

  Это касается и класса, и теста.

  Почему возникла ошибка? Достаточно ли читабелен сам код?

  Часто из-за добавленного нового теста возникает ненужное дублирование, которое проще исправить сейчас, а не ждать, пока оно разойдется по другим тестам.

Тесты должны быть простыми и понятными

Тесты, как уже упоминалось, это тоже код. Он должен быть как можно проще и понятнее. Если не рефакторить и не улучшать тесты, они превратятся в кашу, и все их преимущества сойдут на нет.

Тесты не должны зависеть друг от друга

Тесты должны быть максимально независимыми. У каждого теста должна быть независимая подготовка окружения и данных. Таким образом будет тестироваться только нужный функционал без побочных эффектов.

Тесты должны быть сгруппированы

У каждого тестового случая (не путать с тестом на класс) должна быть своя область видимости для избежания влияния тестов друг на друга. Очень часто приходится видеть подобные тесты:

```
my $foo = Foo->new(name => 'Name'
    , last_name => 'Last Name');
ok($foo);
is($foo->name, 'Name');
is($foo->last_name, 'Last Name');
```

Гораздо лучше написать тест следующим образом:

```
subtest 'create new object' =>
    sub {
     my $foo = Foo->new;

     ok($foo);
};
```

```
6
7  subtest 'correcly initialize
       object' => sub {
8       my $foo = Foo->new(name => '
           Name', last_name => 'Last
           Name');
9
10      is($foo->name, 'Name');
11      is($foo->last_name, 'Last
           Name');
12 };
```

Таким образом четко видно, какой тестовый случай что покрывает, легко вносить изменения и добавлять новые тесты.

Один тест — одна проверка

Каждый тестовый случай должен проверять единственную функциональность.

Таким образом никогда не возникнет тестов, которые тестируют “все”, обычно они называются `test_general`, `test_ok` и тому подобное. В предыдущем примере четко видно, что каждый субтест тестирует конкретную возможность класса.

3A, AAA, Arrange-Act-Assert

По-русски это звучит примерно как Подготовка, Выполнение, Проверка. Тесты должны быть организованы в таком порядке:

- подготовить данные, объект и тому подобное для тестирования;
- вызвать на объекте нужный метод или каким-то образом запустить тестируемый функционал;

- проверить, что он отработал правильно.

```
subtest 'correcly
    concatenates two strings'
     => sub {

    # Подготовка
    my $concatenator =
       Concatenator->new;

    # Выполнение
    my $result =
       $concatenator->cat('
       foo', 'bar');

    # Проверка
    is $result, 'foobar';
};
```

Это делает тесты проще. Легко видеть что тестируется, в каких условиях.

Тестировать поведение, а не конкретную реализацию

Для избежания хрупких тестов, когда малейшее изменение в коде ломает существующие тесты, рекомендуется писать тесты на поведение. Например:

```
eval { $object->die_hard };
is "$@", "We died here for the good reason. Error 42";
```

В данном случае малейшее изменение текста ошибки приведет к сломанному тесту. В нашем случае, мы тестируем, что в тексте ошибки должно присутствовать `Error <код-ошибки>`.

```
eval { $object->die_hard };
like "$@", qr/\s* Error \s+ \d+$/xms;
```

Не увлекаться mock-объектами

Mock-объекты — это объекты, которые можно использовать вместо настоящих объектов, когда их создание невозможно, сложно или они еще не реализованы.

Сами mock-объекты заслуживают своей отдельной статьи, здесь лишь скажем, что не стоит их использовать повсеместно. Основной недостаток mock-объектов проявляется при изменений поведения или интерфейсов настоящих объектов. В данном случае ваши старые тесты будут проходить и дальше, даже несмотря на то, что вы поменяли метод `foo` на `bar` в настоящем классе.

Рекомендуется использовать mock-объекты в режиме подстановки отдельных методов на настоящих объектах, например, используя `Test::MonkeyMock`:

```
my $mocked_url_fetcher = Test::
    MockObject->new;
$mocked_url_fetcher->mock(request
     => sub { 'OK' });

my $real_object = RealObject->new
    ;
$real_object = Test::MonkeyMock->
    new($real_object);
$real_object->mock(
    _build_url_fetcher => sub {
    $mocked_url_fetcher });
```

Таким образом, мы подменили фабричный метод, который создавал объект для получения файла по URL на свой тестовый.

Вместо фабричных методов, конечно, можно использовать и инъекцию зависимостей (если она поддерживается):

```
my $mocked_url_fetcher = Test::
    MockObject->new;
```

```
$mocked_url_fetcher->mock(request
    => sub { 'OK' });

my $real_object = RealObject->new
    (url_fetcher =>
    $mocked_url_fetcher);
```

Перенос создания объектов в фабричные методы

В тестах, как и в коде, рекомендуется использовать фабрики или фабричные методы для создания тестируемых объектов. Это упрощает их инициализацию, избавляет от дублирования. Для тестов, которые используют наследование (например, на основе `Test::Unit` или `Test::Class`) эти методы позволяют подставлять нужные объекты в тесты. Например:

```
package TestBase;
use base 'Test::Unit::TestCase';

sub test_create_timer {
    my $self = shift;

    my $timer = $self->
       _build_timer;

    $self->assert($timer);
}

sub test_increment_time {
    my $self = shift;

    my $timer = $self->
       _build_timer;

    my $old_time = $timer->time;

    $timer->tick;

    my $new_time = $timer->time;
```

```
22
23      $self->assert($new_time >
            $old_time);
24  }
25
26  package OldTimerTest;
27  use base 'TestBase';
28
29  use OldTimer;
30
31  sub _build_timer {
32      my $self = shift;
33
34      return OldTimer->new;
35  }
36
37  package NewTimerTest;
38  use base 'TestBase';
39
40  use NewTimer;
41
42  sub test_some_new_functionality {
43      my $self = shift;
```

```
44
45      my $timer = $self->
            _build_timer;
46
47      $self->assert($timer->
            new_tick);
48  }
49
50  sub _build_timer {
51      my $self = shift;
52
53      return NewTimer->new;
54  }
```

Таким образом, тестируется и соответствие нового класса старому интерфейсу и новый функционал.

Перенос фикстур в отдельные классы

При тестировании большого проекта возникает необходимость создания типовых объектов практически в каждом тесте. Чтобы избежать дублирования, создаются специальные классы-фабрики для создания и инициализации этих объектов. Сама реализация этих фабрик может отличаться. Это может быть один класс для создания всех объектов или же разные классы для создания разных групп объектов. В английской терминологии это Mother или God Object и Test Data Builder.

Использование первого:

```
subtest 'table is round' => sub {
    my $table = MotherObject->
        createTable();

    ok $table->is_round;
```

```
};
```

Использование второго:

```
subtest 'table is round' => sub {
    my $table = TableBuilder->
        create();

    ok $table->is_round;
};
```

Простота важнее абстракций

Не всегда стоит стремиться полностью убрать дублирование из тестов. Излишняя абстракция делает их сложными для восприятия. Например, не стоит всю инициализацию объекта прятать в фабрику фикстур (см. предыдущий раздел). В самом тесте должно быть понятно, как создается

объект, какие атрибуты ему передаются, что именно тестируется.

Юнит-тестирования недостаточно

Тестирование не останавливается на юнит-тестах. Юнит-тесты не могут протестировать систему целиком, нет уверенности, что классы правильно взаимодействуют между собой. В нашей практике мы используем функциональные тесты, которые тестируют систему как черный ящик. Безусловно, в некоторых случаях функциональные тесты тестируют практически тоже самое, что и юнит-тесты, но то, как они это тестируют, сильно отличается.

Кроме автоматического тестирования не стоит пренебрегать и ручным анализом кода, ручным тестированием особо важ-

ных частей системы на граничные случаи. В своей работе мы также используем сканеры безопасности.

Заключение

Это статья лишь бегло покрывает основы тестирования. Еще многое можно сказать о рекомендуемых модулях, организации тестов, mock-объектах, сложностях тестирования, как убедить коллег писать тесты, как убедить заказчика в необходимости рефакторинга и тестов. Пожалуйста, оставляйте свои комментарии, если и дальше хотите видеть в журнале статьи подобной тематики.

■ *Вячеслав Тихановский*

3 Pjam — сервер сборки перловых приложений

Автором представлена собственная разработка для сборки приложений, написанных на языке Perl

Здравствуйте, меня зовут Алексей Мележик. В этой статье я хочу рассказать о pjam — сервере сборки приложений, написанных на Perl.

Немного о себе — я работаю devops-разработчиком в компании, в которой существует множество проектов, написанных на Perl — несколько десятков различных приложений.

Какое-то время я искал готовые решения для сборщиков Perl-приложений промыш-

ленного применения, но не был удовлетворен, по разным причинам, полученными результатами.

Однажды я узнал о разработке под названием pinto и поближе познакомился с этим продуктом (о нем в том числе пойдет речь в данной статье). Естественным образом ко мне пришла идея создания сборочного сервера на базе pinto, так появился pjam. Я являюсь автором данного продукта.

Написание этой статьи имеет под собой цели познакомить perl сообщество с новым продуктом, который, я надеюсь, поможет упростить процессы деплоймента Perl-приложений, а также получить вопросы, пожелания и конструктивную критику, которая, в свою очередь, поможет сделать pjam еще более удобным и полезным для конечного пользователя.

Итак, знакомьтесь — pjam.

Один раз собери — семь раз поставь

Немного предыстории.

Идея сборки приложений на выделенном сервере с последующей установкой на целевых машинах известна давно, это методика чаще всего применяется для ПО, написанного на языках C++ или Java. При данном подходе установка приложений происходит в два этапа. Первый — компиляция программы из исходных кодов на специальном сборочном сервере и получение так называемого дистрибутива, второй — собственно деплоймент (выкладка) готового дистрибутива и его конфигурация на

целевых серверах. Преимуществом такого подхода является возможность многократно устанавливать единожды собранный дистрибутив на машинах с одинаковым окружением. Платой за такую архитектуру является необходимость поддержки окружения сервера, на котором происходит сборка, в соответствии с окружениями устанавливаемых серверов. Однако, такая задача вполне решаема при применении современных методов системного администрирования.

Таким образом, данный способ сборки и установки приложений может быть применен также для приложений, написанных на скриптовых языках, таких как Perl или Python.

Yet another build server

Итак, pjam — сервер сборки перловых приложений. Упрощенная схема работы сервера состоит в следующем: исходные коды забираются из системы контроля версий, компилируются вместе с зависимостями и упаковываются в единый архив. В итоге мы получаем дистрибутив, готовый к установке на соответствующих целевых машинах. Фактически все, что нужно админу для того, чтобы запустить приложение (я упускаю стадию конфигурации и прочие детали) — это закачать архив, распаковать его, добавить в `PERL5LIB` библиотеки, собранные в дистрибутиве:

```
wget http://your.pjam.server/
    projects/1/builds/273/
    artefacts/app.tar.gz && tar -
    xzf app.tar.gz && export
    PERL5LIB=app/cpanlib/lib/perl5
```

Здесь `cpanlib/` — директория внутри дистрибутива, в которую установлены все зависимости.

Сборка и борьба с зависимостями

Описание принципов работы pjam хочется начать с описания ядра его функциональности, а именно — процесса сборки дистрибутива из исходных кодов. Самое сложное и неприятное в этом процессе, с чем сталкивается любой build-инженер, причем не важно, идет речь о сборке Java-приложений или любых других (Perl здесь не исключение), — это *зависимости.* Для управления зависимостями pjam использует pinto — перспективная разработка, уже используемая многими Perl-разработчиками для деплоймента приложений и управления

CPAN-репозиториями. Но прежде чем говорить о специфике использовании pinto в pjam, немного расскажу о том, какие именно зависимости бывают в pjam-проектах.

CPAN-модули

Итак, в процессе создания сборок pjam также сталкивается с проблемой разрешения зависимостей. Под зависимостями здесь понимается два типа сущностей: первый тип зависимостей — уже известные многим Perl-программистам CPAN-модули. Как правило, такие модули лежат либо в публичных, либо в приватных CPAN-репозиториях. Pjam можно настроить на использование одного или нескольких таких CPAN-репозиториев, чтобы он «знал»,

откуда брать CPAN-модули[1].

Проекты и компоненты

Второй тип зависимостей, обрабатываемых в pjam, — это части приложения, расположенные в виде исходного когда в системе контроля версий[2]. В предметной области pjam такие части называются

---

[1]Выделение кода в CPAN-модули широко практикуется во многих компаниях, ведущих разработку на перле. Например, в моей компании десятки модулей, используемые в приложениях, выложены в приватный CPAN. Однако, хочу сразу заметить, что хотя CPAN и CPAN-репозитории являются важной составляющей деплоймента и дистрибуции Perl-приложений, детальное обсуждение этой темы выходит за рамки данной статьи.

[2]На данный момент pjam поддерживает только subversion, но в будущем автор может добавить поддержку и других систем контроля версий.

компонентами. Очень часто, особенно в больших приложениях, удобно делить исходный код приложения на отдельные куски и размещать их по разным ресурсам системы контроля версий. В случае с subversion это могут быть отдельные проекты и/или отдельные директории одного проекта в репозитории, в случае с git — отдельные git-репозитории. Таким образом, в контексте pjam каждое собираемое приложение является *проектом*, который в свою очередь содержит упорядоченный список компонентов, каждый из которых представлен свои ресурсом в системе контроля версий. В процессе сборки pjam проходит по данному списку компонент за компонентом и делает сборку всего проекта. Минимальным требованием к содержимому компонента является наличие в его исходном каталоге в системе контроля версий правильного сборочного файла

формата `Build.PL` или `Makefile.PL`[3].

Как происходит сборка

Разрешение зависимостей

В процессе сборки исходный код каждого компонента получается из системы контроля версий, и для него запускается стандартный цикл команд, превращающий исходник компонента в дистрибутив[4]:

---

[3]Данный файл определяет процесс компиляции исходного кода, а также указывает на зависимости, которые требуются данному компоненту. Существует несколько известных перловых пакетов, с помощью которых можно создавать подобные файлы, например, Module::Build или ExtUtils::MakeMaker.

[4]К сожалению, термин дистрибутив явно перегружен при употреблении в технической литерату-

```
<получить компонент из системы
    контроля версий> && perl Build
    .PL|Makefile.PL && ./Build|
    make manifest && ./Build|make
    dist  && <добавить дистрибутив
     компонента в локальный
    репозиторий>
```

Полученный дистрибутив посредством pinto (система управления CPAN-репозиторием) добавляется в локальный pinto-репозиторий, при этом *все* зависимости (а также зависимости зависимостей, т.е. рекурсивно), объявленные в сборочном файле компонента также добавляются в репозиторий. В итоге компонент «превращается» в обычный CPAN-модуль, помещенный в

ре на данный момент. Под ним может пониматься в зависимости от контекста: дистрибутив всего приложения (об этом уже говорилось); дистрибутив CPAN-модуля (архив, загружаемый с CPAN-зеркала); любой исходный код, упакованный в архив.

локальный репозиторий.

Фазу создания дистрибутивов для всех компонентов приложения и добавление этих дистрибутивов в репозиторий можно назвать *pinto*-фазой. По завершению данной фазы мы получаем pinto-репозиторий с собранными в нем всеми зависимостями нашего приложения. Для более глубокого понимания pinto-репозиториев можно обратиться к документации pinto, но для нас лишь важно, что на этом этапе, если все проходит успешно, мы имеем полный набор зависимостей, собранных в одном месте в файловой системе. И что еще более важно, теперь можно установить все зависимости из локального pinto-репозитория как обычные CPAN-модули. Переходим к фазе компиляции.

Фаза компиляции

Второй фазой сборки проекта в pjam является собственно компиляция всех собранных зависимостей, уже находящихся в локальном репозитории[5]. Делается это очень просто. Дистрибутив каждого компонента устанавливается в локальную директорию сборки с помощью того же pinto:

```
1 pinto ––root <локальный pinto–
    репозиторий> install –s <стек
    для сборки> –l '<локальная
    директория сборки>'
```

По окончанию установки всех компонентов локальная директория сборки упаковывается в архив. Получаем готовый

[5]pinto-репозиторий является одновременно сущностью сугубо специфической для pinto, но в тоже время имеет интерфейс CPAN-репозитория.

дистрибутив.

Обработка ошибок

Конечно же следует упомянуть о том, что на любой из фаз сборки могут возникнуть ошибки, начиная от недоступности системы контроля версий, неопределенными зависимостями, до ошибок компиляции со сторонними библиотеками или провалом модульных тестов, в этом случае pjam прерывает процесс сборки и выводит соответствующее сообщение в лог сборки. Детализация лога может быть также настроена в самом проекте.

Инкрементальные сборки

Одной из замечательных особенностей pjam-сервера является технология «инкрементальных» сборок. Процедура сборки, описанная выше, может повторяться из раза в раз, для одного и того же проекта, отражая тем самым изменения в коде, вносимые разработчиками и фиксирующими их в системе контроля версий. При инкрементальном сборочном процессе очередная сборка «наследует» состояние предыдущей в виде:

- состояния локального pinto-репозитория на момент завершения предыдущей сборки;
- локальной директории предыдущей сборки.

Подобное сохранение состояния предыдущих сборок существенно ускоряет процесс создания новой сборки — нет необходимости устанавливать каждый раз все зависимости с нуля, ведь часть зависимостей уже была поставлена ранее.

Что нам дает pinto?

Сама идея инкрементальных сборок не нова, но вся соль в том, что на низком уровне pjam используется pinto для сохранения состояния предыдущих сборок. Когда запускается очередная сборка, происходит следующие:

- локальная сборочная директория копируется в новую сборку;

- pinto-стек[6], отражающий состояние репозитория на момент завершения предыдущей сборки, копируется в новый стек очередной сборки;
- запускается сборочный процесс.

Не вдаваясь в специфику pinto, это означает, что при данной архитектуре возможно следующее:

---

[6]pinto-стек — это выборка подмножества версий модулей, лежащих в репозитории (CPAN index view).

Стеки — очень мощное средство, позволяющее, например, всегда ставить требуемые версии модулей. Приближенной аналогией стеков могут быть ветви исходного кода в системе контроля версий. Стеки позволяют иметь несколько подмножеств CPAN-индексов в одном локальном репозитории. Подход, применяемый в pjam, — это создание нового стека под каждую сборку. Фактически это равноценно созданию мгновенного снимка репозитория для конкретной сборки.

- Видеть, что именно обновилось в очередной сборке. Получается элементарным сравнением стеков предыдущей и текущей сборки.

```
pinto diff build-previous-stack-id build-new-stack-id
```

- Сравнивать две различные сборки. Получается элементарным сравнением стеков сборок.

- Откатить состояние проекта до требуемой сборки, включая состояние локального репозитория и стека. Достигается просто копированием стека сборки, к которой требуется осуществить откат, в новую сборку.

```
pinto new build-old-stack-id build-new-stack-id
```

Все операции со стеками достаточно дешевые (не требуют много времени для выполнения) и, что важно, уже реализованы в самом pinto, остается только правильно использовать в самом pjam.

Асинхронные сборки

Немного расскажу о том, как создаются сборки с точки зрения конечного пользователя pjam-интерфейса. Любая сборка, конечно же, занимает немалое время. Что бы не заставлять пользователя ждать, pjam использует технологию обработки асинхронных задач. Это означает, что сборки не выполняются мгновенно, а ставятся в очередь и затем обрабатываются асинхронным шедулером (`delayed_job`). Для пользователя интерфейса это означает, что

ему не нужно ждать, пока сборка закончится, он добавляет сборку в очередь и по ее окончанию получает соответствующее уведомление через jabber или видит ее обновленный статус в интерфейсе.

Параллельные сборки

К сожалению, из-за специфики pinto (использование файловых блокировок при работе с репозиторием), выполнение сборок из очереди происходит последовательно. Фактически, параллельное выполнение сборок невозможно, хотя и технически реализуемо в самом шедулере.

Я общался с автором pinto на данную тему, в следующих релизах pinto он обещал поменять архитектуру файловых блокировок,

что возможно поможет решить данную проблему.

Yet Another CI-сервер?

Отличия и схожести с каноническим сервером непрерывной интеграции.

Начнем с перечисления тех особенностей, которые любой CI-сервер должен предоставлять, и которые есть в pjam.

- Интеграция с системой контроля версий — есть, но пока только для subversion. В будущем, возможно, будет добавлена поддержка git.
- Уведомление о статусах сборок — происходит посредством jabber-клиента, настройки jabber-аккаунта и

jabber-сервера задаются на странице конфигурации pjam. Каждый проект имеет уникальный список пользователей, которым будет приходить рассылка уведомлений.

- Работа со сборками и артефактами — каждая успешная сборка в pjam порождает архив (т.н. артефакт), который может быть загружен для деплоймента. Есть функции удаления, «заморозки» сборки (защита от случайного удаления), к сборке можно добавить описание или присвоить статус релиза. Есть очень удобная функция загрузки дистрибутива последней успешной сборки в проекте.

Чего нет в pjam (по сравнению с тем же Jenkins).

- Вызова сторонних задач (remote hooks) по факту завершения сборки. Функция не то что бы очень необходимая, пока не уверен, что хочу добавлять ее.
- Ротации сборок.
- Аутентификации. Хотя действия пользователей логируются и отображаются через интерфейс, всегда можно увидеть, с какого хоста (делается попытка преобразования ip-адресов в имена хостов) что и когда было изменено в конфигурации проекта, а также кто запустил сборку.
- Автоматического опроса системы контроля версий на появление новых коммитов и запуска сборок по данному событию. Сборки инициируются явно через интерфейс, возможен запуск сборок в стиле RESTful API обычными клиентами curl или wget.

В общем и целом pjam не является многофункциональным CI-сервером, как тот же Jenkins, хотя имеет минимальный набор функций, позволяющий использовать его в процессах непрерывной интеграции. Я лично делаю сборки в pjam, а деплоймент совершаю посредством Jenkins, загружая дистрибутивы последних успешных сборок из pjam.

Документация по установке и дистрибутив pjam

Документация находится на странице проекта на GitHub — https://github.com/melezhik/pjam-on-rails. На данный момент pjam ставится получением кода из GitHub и далее настраивается и запускается как стандартное Rails-приложение. В будущем

я рассматриваю вариант более конвекционной дистрибуции.

Откуда я взял идеи для pjam

- pinto
- jenkins
- Reliable Software Releases through Build, Test, and Deployment Automation Jez Humble, David Farley
- delayed_job

Почему pjam написан на Ruby?

Обычно это первый вопрос, который задают люди услышав о pjam. На самом деле pjam — это всего лишь обертка вокруг pinto,

ядро системы реализуется именно в нем. Pjam предоставляет интерфейс к сборщику. А так это web-интерфейс, мне было проще написать на Ruby on rails. Ну… и как сказал мой коллега yakudza — теперь осталось переписать pjam с Ruby на Perl (:

Заключение

Если вас заинтересовал данный продукт, вы можете легко установить его и начать им пользоваться. Pjam все еще находится в стадии разработки, без стабильного релиза, но тем не менее большинство функций оттестировано и их поведение достаточно стабильно. Ну и конечно, всегда есть возможность сообщить об ошибках или поучаствовать в разработке, заходите на https://github.com/melezhik/pjam-on-rails.

■ *Алексей Мележик*

4 Атрибуты в Perl

Рассмотрен механизм атрибутов

В предыдущем номере я слегка затронул атрибуты и получил несколько отзывов с просьбами сделать статью, которая будет посвящена этому механизму. Так и появилась эта статья. Тем, кто использует `mod_perl`, я советую посмотреть часть статьи «Подводные камни», прежде чем изучать данный функционал.

Немного теории

Атрибуты были введены в Perl, если мне не изменяет память, в версии 5.6 как экспериментальный механизм. Более того, они и остаются экспериментальным механиз-

мом, потому их поведение может меняться от версии к версии perl.

Основная идея атрибутов — передача состояния функциям, добавление дополнительного поведения к переменным и функциям.

goto

Прежде чем продолжать, я бы хотел напомнить о существовании такой штуки как `goto`. Эта штука является очень проблемной, потому `goto` заслужил репутацию оператора для спагетти кода.

В Perl есть три формы оператора `goto`:

```
goto LABEL

```

```
goto EXPR

goto &NAME
```

Нас интересует только третья форма. Данная форма оператора `goto` называется «специальная форма goto». И это единственная форма оператора, которую можно использовать, и использование которой не является дурным тоном.

Эта форма отличается тем, что при ее использовании происходит JUMP, но не вызов процедуры, что существенно быстрее. Данный прием нашел широкое применение в разработке модулей, которые должны обрабатывать вызов несуществующих методов, в обертках, когда нужно вместо одного метода вызывать другой. В первом случае это делается при помощи AUTOLOAD. Но хочу предостеречь.

Это использование goto действительно оправдано, но злоупотреблять им не стоит, потому что в результате можно получить совершенно несопровождаемый код.

В случае с `AnyEvent` такой код называют “callback hell”, а в случае с goto “goto hell”, соответственно.

Атрибуты непосредственно

Стандартный механизм работы с атрибутами в Perl печален. Проблема в том, что он несколько громоздкий, странный, не очень понятный и, к тому же, выглядит весьма корявым.

Именно по этой причине в поставку Perl входит модуль `Attribute::Handlers`,

который существенно упрощает работу с ними и добавляет много полезностей.

Мы рассмотрим этот модуль далее, но сначала я предлагаю разобраться, как же они работают на уровне языка. `Attribute::Handlers`, по сути, является оберткой над стандартным механизмом атрибутов.

Стандартные атрибуты

В стандартную поставку Perl входят: `method`, `locked`, `lvalue` и еще атрибуты для работы с потоками.

В данный момент нас интересует только `lvalue`-атрибут.

Например, у нас есть код, который выглядит следующим образом:

```
my $val;
sub canmod : lvalue {
    $val;  # or:  return $val;
}
sub nomod {
    $val;
}
canmod() = 5;   # assigns to $val
nomod()  = 5;   # ERROR
```

Код взят из документации perl, но он четко показывает применение данного атрибута. Но сама документация говорит, что `lvalue`-функции являются экспериментальными, а потому их синтаксис и поведение может поменяться в следующих версиях Perl.

Атрибуты, к тому же, могут использоваться не только с функциями, но и с переменными.

Атрибуты, связанные с многопоточностью, я рассматривать не буду, ибо ценность этих знаний стремится к нулю.

Стандартные атрибуты это конечно хорошо, но гораздо интереснее создавать пользовательские атрибуты. Для того, чтобы это сделать, надо разобраться как они работают вообще.

Под капотом

Внутри атрибуты устроены следующим образом.

Когда во время компиляции perl встречает атрибут, он пытается сделать следующий вызов:

```
1 __PACKAGE__—>
```

```
MODIFY_CODE_ATTRIBUTES(\&mySub
, @list_of_attributes);
```

Этот код выполняется на стадии BEGIN, но есть исключения, например, функция MODIFY_SCALAR_ATTRIBUTES выполняется при инициализации переменной.

Для того, чтобы добавить атрибуты, необходимо написать функцию в своем модуле, которая будет называться MODIFY_*_ATTRIBUTES, где вместо звездочки — желаемый тип.

Также есть весьма полезный модуль attributes, у которого есть функция get. Эта функция при вызове обращается к обработчику, который имеет название FETCH_CODE_ATTRIBUTES, для атрибутов функций. Вообще, атрибуты весьма интересная вещь. Я встречал несколько

оправданных ее применений:

- В `Catalyst` так проверяется, авторизован пользователь или нет.
- Абстрактные классы, private-, public-, protected-переменные или методы при помощи атрибутов делаются на раз-два.
- Экспортирование функций тоже весьма интересно делается на атрибутах.

Их очень полезно также использовать для модификации стандартного поведения. Возьмем, к примеру, `tie`, что позволяет изменять стандартное поведение стандартных структур данных весьма нетривиальным образом. Атрибуты делают то же самое, но выгодно отличаются тем, что изменение поведения атрибутами еще более нетривиально, а при неправильном

использовании отладка кода превращается в сущий ад.

Для того, чтобы атрибут был выполнен успешно, он должен возвращать пустой список, например:

```
return;
```

Если же будет возвращено нечто другое, то это приведет к ошибке.

Позволю себе напомнить, как работает `BEGIN` в Perl.

```
#!/usr/bin/env perl
use strict;

print "After begin";

BEGIN {
    print "I AM AT BEGIN\n";
}
```

Результат работы этой программы:

```
I AM AT BEGIN
After begin%
```

Далее примеры кода, которые иллюстрируют вышесказанное.

Напишем свой первый атрибут

```
#!/usr/bin/env perl
use strict;
use warnings;
use Data::Dumper;

print "After begin\n";

mysub();

print 'Before $x init', "\n";
my $x : Hello = 1;
```

```
12 print 'After $x init', "\n";
13 print '$x: ', $x, "\n";
14
15 BEGIN {
16     print "I AM AT BEGIN\n";
17 }
18
19 sub MODIFY_CODE_ATTRIBUTES {
20     print "I am code attributes
          modifier!\n";
21     print Dumper \@_;
22     return;
23 }
24
25 sub MODIFY_SCALAR_ATTRIBUTES {
26     print "I am scalar attributes
          modifier!\n";
27     print Dumper \@_;
28     return;
29 }
30
31 sub mysub : Hello {
32     print "YAPH!\n";
```

```
33 }
```

Обратите внимание, что абсолютно все равно, как называются атрибуты.

Это программа работает следующим образом:

1. Сначала выполняется `BEGIN`-блок.
2. Затем выполняется `MODIFY_CODE_ATTRI`, т.к. мы добавили к функции `mysub` атрибут `Hello`. Атрибуты, как мы помним, срабатывают на `BEGIN`-стадии. Кстати, обратите внимание на то, что встроенные атрибуты начинаются с маленькой буквы, тогда как пользовательские атрибуты документация советует называть с большой буквы.
3. Затем программа напечатает `After begin`, после чего будет исполнена

функция mysub.

4. Затем опять print, после которого будет инициализация переменной $x. А вот так выглядят атрибуты переменных. Например, переменная $x имеет такой же атрибут — Hello, но этот атрибут будет выполнен при инициализации переменной. Затем будет проинициализирована переменная и вызвана MODIFY_SCALAR_ATTRIBUTES.

Вывод программы будет выглядеть таким образом:

```
I AM AT BEGIN
I am code attributes modifier!
$VAR1 = [
          'main',
          sub { "DUMMY" },
          'Hello'
        ];
```

```
After begin
YAPH!
Before $x init
I am scalar attributes modifier!
$VAR1 = [
          'main',
          \undef,
          'Hello'
        ];
After $x init
$x: 1
```

Функция MODIFY_CODE_ATTRIBUTES вызывается примерно следующим образом:

```
__PACKAGE__->
    MODIFY_CODE_ATTRIBUTES(\
    $subref, @attrs);
```

Где attrs — список атрибутов. Например, мы можем добавить нашей функции mysub еще один атрибут, тогда её код будет выгля-

деть следующим образом:

```
sub mysub : Hello : World {
    print "YAPH!\n";
}
```

А вся программа, соответственно, будет выглядеть:

```
#!/usr/bin/env perl
use strict;
use warnings;
use Data::Dumper;

print "After begin\n";

mysub();

print 'Before $x init', "\n";
my $x : Hello = 1;
print 'After $x init', "\n";
print '$x: ', $x, "\n";

BEGIN {
```

```
16      print "I AM AT BEGIN\n";
17  }
18
19  sub MODIFY_CODE_ATTRIBUTES {
20      print "I am code attributes
            modifier!\n";
21      print Dumper \@_;
22      return;
23  }
24
25  sub MODIFY_SCALAR_ATTRIBUTES {
26      print "I am scalar attributes
            modifier!\n";
27      print Dumper \@_;
28      return;
29  }
30
31  sub mysub : Hello : World {
32      print "YAPH!\n";
33  }
```

А ее вывод будет немного отличаться. В

частности, вызов MODIFY_CODE_ATTRIBUTES будет выглядеть вот так:

```
__PACKAGE__->
    MODIFY_CODE_ATTRIBUTES(\
    $mysubref, 'Hello', 'World');
```

А сам вывод так:

```
I AM AT BEGIN
I am code attributes modifier!
$VAR1 = [
           'main',
           sub { "DUMMY" },
           'Hello',
           'World'
         ];
After begin
YAPH!
Before $x init
I am scalar attributes modifier!
$VAR1 = [
           'main',
           \undef,
```

```
            'Hello'
        ];
After $x init
$x: 1
```

А вот пример кода, когда атрибут возвращает непустой список. Это приведет к ошибке:

```
#!/usr/bin/env perl
use strict;
use warnings;
use Data::Dumper;

mysub();

sub MODIFY_CODE_ATTRIBUTES {
    print "I am code attributes
       modifier!\n";
    print Dumper \@_;
    return ('Hello', 'World');
}

```

```
14 sub mysub : Hello : World {
15     print "YAPH!\n";
16 }
```

А вот и сама ошибка:

```
1 I am code attributes modifier!
2 $VAR1 = [
3           'main',
4           sub { "DUMMY" },
5           'Hello',
6           'World'
7         ];
8 Invalid CODE attributes: Hello :
    World at attrs.pl line 18.
9 BEGIN failed--compilation aborted
    at attrs.pl line 18.
```

С особенностями работы атрибутов на низком уровне можно ознакомиться в соответствующем разделе `perldoc`.

use attributes;

Помимо `MODIFY_*_ATTRIBUTES`, у пакета могут быть представлены методы другого типа — `FETCH_*_ATTRIBUTES`.

`use attributes;` дает нам доступ к двум методам, `get` и `reftype`. Принцип работы `get` можно проиллюстрировать следующим примером кода:

```
#!/usr/bin/env perl
use strict;
use warnings;
use Data::Dumper;
use attributes qw/get/;

mysub();

my @al =  attributes::get(\&mysub
    );
print Dumper \@al;
```

```
11
12 sub MODIFY_CODE_ATTRIBUTES {
13     print "I am code attributes
          modifier!\n";
14     print Dumper \@_;
15     return;
16 }
17
18 sub FETCH_CODE_ATTRIBUTES {
19     print "FETCH_CODE_ATTRIBUTES
          called!\n";
20     print Dumper \@_;
21     return ('Hello', 'World');
22 }
23 sub mysub : Hello : World {
24     print "YAPH!\n";
25 }
```

Вывод программы будет выглядеть следующим образом:

```
1 I am code attributes modifier!
2 $VAR1 = [
```

```
3             'main',
4             sub { "DUMMY" },
5             'Hello',
6             'World'
7           ];
8 YAPH!
9 FETCH_CODE_ATTRIBUTES called!
10 $VAR1 = [
11             'main',
12             sub { "DUMMY" }
13           ];
14 $VAR1 = [
15             'Hello',
16             'World'
17           ];
```

Мы можем видеть, что функция вернула нам то, что вернула `FETCH_CODE_ATTRIBUTES`.

Функция `reftype` интереснее. Всем известно, что в Perl есть такая функция как `ref`. Ра-

ботает она примерно следующим образом:

```
ref {};
```

вернет нам HASH, т.к. это есть ссылка на хеш. Не забываем, что `reftype` по умолчанию не импортируется, потому мы должны писать `use attributes qw/reftype/;`. Предлагаю вашему вниманию следующий кусок кода.

```
ref {} eq reftype {} and print "
    EQUIVALENT!";
```

Этот код напечатает EQUIVALENT!

Т.е. в данном случае, получается, что две функции работают одинаково. Однако, Perl не PHP и эти функции отличаются. Вся штука в том, что `reftype` игнорирует пакет и возвращает примитивный тип данных по ссылке. Приведу пример:

```
#!/usr/bin/env perl
use strict;
use attributes qw/reftype/;
$\ = "\n";

my $hash = {};
my $pack = MyPack—>new();

printf 'ref $pack: %s ref $hash:
    %s%s',
     ref $hash, ref $pack, "\n";

printf 'ref $pack: %s ref $hash:
    %s%s',
     reftype $hash, reftype $pack,
         "\n";

package MyPack;
use strict;

sub new {
    return bless {}, __PACKAGE__;
}
```

```
21
22 1;
```

Этот код напечатает:

```
1 ref $pack: HASH ref $hash: MyPack
2 ref $pack: HASH ref $hash: HASH
```

Вывод

Да, действительно, данный механизм слегка неудобен и громоздок, потому вместо него стоит использовать стандартный модуль Perl для работы с атрибутами — `Attribute::Handlers`, а то, как они работают внутри, необходимо, но не обязательно знать для общего развития.

Attribute::Handlers

Если раньше атрибуты позволяли нам получить несколько параметров и вызов их схематически можно было изобразить как:

```
1 __PACKAGE__->M_C_A($subref,
    @list_of_attrs)
```

то вызов `Attribute::Handlers` куда более практичный и может быть записан так:

```
1 __PACKAGE__->ATTR_HANDLER(
    $package, $symbol, $referent,
    $attr, $data, $phase,
    $filename, $linenum)
```

Информации передается на порядок больше, что есть хорошо и дает нам возможность задавать более осмысленные

атрибуты. Хорошим примером использования атрибутов является `Attribute::Protected`, где на их базе построена реализация инкапсуляции.

Инкапсуляция — один из трех китов ООП, вместе с полиморфизмом и наследованием. Так получилось, что ООП в Perl очень простое, но инкапсуляции как таковой нет.

А самое интересное, что нет способа сделать это проще, чем на атрибутах. Вот весь код модуля:

```
1 package Attribute::Protected;
2
3 use 5.006;
4 use strict;
5 use warnings;
6
7 our $VERSION = '0.03';
8
9 use Attribute::Handlers;
```

```
10
11 sub UNIVERSAL::Protected : ATTR(
       CODE) {
12      my($package, $symbol,
           $referent, $attr, $data,
           $phase) = @_;
13      my $meth = *{$symbol}{NAME};
14      no warnings 'redefine';
15      *{$symbol} = sub {
16      unless (caller->isa($package)
           ) {
17           require Carp;
18           Carp::croak "$meth() is a
                protected method of
              $package!";
19      }
20      goto &$referent;
21      };
22 }
23
24 sub UNIVERSAL::Private : ATTR(
       CODE) {
```

```
25      my($package, $symbol,
           $referent, $attr, $data,
           $phase) = @_;
26      my $meth = *{$symbol}{NAME};
27      no warnings 'redefine';
28      *{$symbol} = sub {
29      unless (caller eq $package) {
30          require Carp;
31          Carp::croak "$meth() is a
                private method of
              $package!";
32      }
33      goto &$referent;
34      };
35  }
36
37  sub UNIVERSAL::Public : ATTR(CODE
        ) {
38      my($package, $symbol,
           $referent, $attr, $data,
           $phase) = @_;
39      # just a mark, do nothing
40  }
```

```
41
42 1;
43 __END__
```

Более того, данный модуль позволяет пробрасывать в обработчики атрибутов дополнительные параметры. Если использовать атрибут примерно так:

```
1 sub mysub : Hello (one, two,
    three) {...};
```

то в этом случае обработчик получит в переменной `$data` ссылку на массив вида `['one', 'two', three]`.

Я описал в начале статьи третью форму `goto` только для того, чтобы ценность данного примера была оценена по достоинству. И для того, чтобы в коде примера `goto` не играл роль красной тряпки для быка, и мне не пришлось бы в комментариях

объяснять, что это ни разу не спагетти код. Спасибо за понимание.

Если приведенный выше код не совсем очевиден, я добавлю в следующую статью описание `typeglob` и стадий исполнения перла.

Подводные камни

Фаза `CHECK` выбрана для обработки атрибутов не зря. В этой стадии таблица символов уже заполнена, и над ней можно издеваться. Однако, как известно, в `mod_perl` этой фазы нет, а потому тем, кто его использует придется городить костыли для использования атрибутов, или не использовать их вообще.

Атрибуты это очень интересная и забав-

ная штука, однако любую технологию необходимо использовать с умом, дабы не получить на выходе приложение, полное архитектурных излишеств. Как говорил герой комиксов: «С большой силой приходит большая ответственность».

До новых встреч.

■ *Дмитрий Шаматрин*

5 Minilla — система подготовки дистрибутивов для CPAN

На сегодняшний день существует множество различных утилит для создания дистрибутивов и публикации их на CPAN. Minilla выделяется в их числе своей простотой и удобством, не превращая простой и рутинный процесс подготовки сборки в карго-культ из нагромождения плагинов и сложных конфигураций.

Введение

Minilla была разработана небезызвестным японским программистом *Tokuhiro Matsuno*. Названа она была по имени одного из японских монстров (*кайдзю*) из серии фильмов о Годзилле. Судя по описанию в википе-

дии — это маленькое и дружелюбное к людям создание, что, вероятно, и отражает характеристики данной утилиты. Любопытно, что первоначально проект был назван *Minya* (одно из прозвищ Minilla), но заметили, что набирать название команды `minya` на QWERTY-клавиатуре не очень-то удобно (особенно сочетание `ny`), в итоге пришли к команде `minil` и имени Minilla.

Основной девиз системы — соглашение вместо конфигурации. Minilla предлагает простой свод правил, с которыми вы можете или согласиться, или найти себе другую систему подготовки дистрибутива:

- Проект ведётся в системе контроля версий Git.
- Список файлов, попадающих в релиз, совпадает с выводом команды `git ls-files`.

- Файлы модуля, написанные на Perl, помещаются в каталог `lib`.
- Исполняемые файлы (скрипты) помещаются в каталог `script`.
- Модуль имеет статический список зависимостей, которые описываются в `cpanfile`.
- Модуль имеет файл `Changes`, описывающий изменения.

Если правила приемлемы, то можно продолжать.

В разработке любого CPAN-дистрибутива можно выделить четыре основных этапа:

1. Создание скелета дистрибутива
2. Разработка
3. Тестирование
4. Релиз дистрибутива на CPAN

Все этапы, кроме разработки, хорошо поддаются автоматизации, поэтому Minilla покрывает их достаточно полно.

Создание нового дистрибутива

Minilla содержит утилиту командной строки `minil`, с помощью которой осуществляются все операции с дистрибутивом. Для создания нового дистрибутива используется команда:

```
$ minil new Dist-Name
```

где *Dist-Name* — это название создаваемого дистрибутива модуля.

В текущей директории создаётся каталог `Dist-Name`, в котором образуется следующая структура каталогов и файлов:

```
$ git ls-files

.gitignore
.travis.yml
Build.PL
Changes
LICENSE
META.json
README.md
cpanfile
lib/Dist/Name.pm
minil.toml
t/00_compile.t
```

При создании файлов Minilla использует информацию из `~/.gitconfig`, в частности, ваше имя и email-адрес для формирования информации об авторе.

Файлы `Build.PL` и `META.json` генерируются автоматически и не подразумевают ручной правки, но они всё равно добав-

ляются в индекс вашего проекта для того, чтобы всегда иметь возможность провести сборку и установку модуля непосредственно из git (например, с помощью `cpanm`). Возможность установки из git — это одно из ключевых декларируемых достоинств Minilla.

Далее необходимо сделать первый коммит (все файлы уже добавлены в индекс):

```
$ git commit -m "initial commit"
```

Подготовка релиза

После написания модуля и тестов подходит момент, когда необходимо сделать релиз. Minilla предполагает, что у вас настроен внешний репозиторий, в который отправляются все изменения. Это может

быть github или какой-либо частный git-репозиторий:

```
# Добавляем внешний репозиторий origin, в который будут отправляться релизы
$ git remote add origin https://github.com/user/Dist-Name.git
```

После чего можно перейти к созданию релиза. Редактируется запись в файле `Changes`, в которой описываются сделанные в релизе изменения, под меткой `{{$NEXT}}` (если вы забудете это сделать, то при релизе minil вежливо попросит вас об этом и откроет ваш любимый `$EDITOR`):

```
{{$NEXT}}

    - my new awesome module
```

Выполняется команда:

```
$ minil release
```

После чего происходит сборка дистрибутива и запрашивается версия нового релиза. Номер версии и текущее время релиза автоматически добавляются в файл `Changes` под меткой. Обновляется значение переменной `$VERSION` в главном модуле и в файле `META.json`. Создаётся новый коммит, который помечается неаннотированным тегом с номером очередной версии релиза в git. Происходит запуск релизных тестов, и, в случае успеха, формируется архив дистрибутива, а текущая ветка master отправляется во внешний репозиторий origin.

В конце процедуры свежий дистрибутив отправляется на CPAN. Для отправки используется модуль `CPAN::Uploader`, который ожидает обнаружить данные

для аутентификации на PAUSE в файле `~/.pause`:

```
user EXAMPLE
password your-secret-password
```

Чтобы не отправлять дистрибутив на CPAN, можно установить переменную окружения `FAKE_RELEASE=1`:

```
$ FAKE_RELEASE=1 minil release
```

Конфигурация

Несмотря на жёсткие правила игры, Minilla всё же допускает возможность конфигурации некоторых аспектов дистрибутива. Для этих целей служит файл `minil.toml` в корне проекта. Файл имеет формат TOML (гибрид INI и JSON). По названиям

опций можно заметить влияние `Module::Install`, например:

```
readme_from = "lib/My/Foo.pod"
```

`readme_from` позволяет задать файл, из которого будет генерироваться `README.md` (по умолчанию, это главный модуль).

Следующий пример конфигурации позволяет не загружать дистрибутивы на CPAN при создании релиза:

```
[release]
do_not_upload_to_cpan=true
```

Полный список опций конфигурации доступен в документации Minilla. Но подразумевается, что необходимость конфигурации — это редкое исключение из правил.

Миграция

Minilla содержит базовую поддержку миграции существующего проекта, для чего необходимо выполнить команду в корне этого проекта:

```
$ minil migrate
```

Minilla сделает простой анализ и инициирует git-репозиторий (если до этого не использовался git), попытается сформировать файл `META.json` для того, чтобы сгенерировать из него корректный `cpanfile`. Создаст файлы `Changes` и `LICENSE` (если их не было), адаптирует `Changes` в формат, принятый в Minilla. Будут удалены некоторые уже бесполезные файлы, такие как `MANIFEST*`, `Makefile.PL`, `README`, `dist.ini` и некоторые другие. В случае наличия `dist.ini` попытается выпол-

нить миграцию конфигурации в файл `minil.toml`.

Заключение

Minilla показывает нам подход, в котором процедура релиза — это быстрый и несложный процесс, который не должен быть отягощён часами подготовки всех служебных файлов, обновления списков файлов и версий — одна команда, и готово, без лишних телодвижений и ничего не пропущено. Формат репозитория удобен для совместной работы, поскольку правила просты и жёстко зафиксированы, не требуется время для изучения особенностей сборки. Модуль готов к установке прямо из git, что также удобно как для совместной разработки, так и для быстрого

тестирования новых версий.

Кстати, пользователи `Dist::Zilla`, которым понравились идеи Minilla, но которые не хотят пока отказываться от привычного инструмента, могут установить родственную систему Milla, которая следует всем правилам Minilla, но работает поверх dzil.

■ *Владимир Леттиев*

6 Обзор CPAN за март 2014 г.

Рубрика с обзором интересных новинок CPAN за прошедший месяц.

Статистика

- Новых дистрибутивов — 224
- Новых выпусков — 844

Новые модули

- Module::Spy

Модуль `Module::Spy` позволяет отслеживать вызовы методов заданного клас-

са/объекта. Это может быть полезно при mock-тестировании, когда требуется проверить, что происходит вызов метода, но при этом не проводить реального запуска, возвращая заранее подготовленный результат.

- Nginx::FastCGI::Cache

Если вы используете кэширование ответов FastCGI-приложений в nginx и вам требуется вручную управлять кэшем nginx, то модуль `Nginx::FastCGI::Cache` даст вам такую возможность. По заданному пути и ключам кэширования модуль может как полностью очищать кэш, так и очищать файлы для заданного URL.

- Log::Journald

`Log::Journald` позволяет отправлять сообщения в журнал systemd. В отличие от классического syslog, в журнал systemd можно передавать структурированные сообщения, бинарные данные и UTF-8 строки. Дистрибутив также содержит бэкенды для популярных модулей `Log::Dispatch` и `Log::Log4perl`.

- AnyEvent::LeapMotion

Технология LeapMotion для захвата движения с разрешением до сотых долей миллиметра теперь доступна в Perl-приложениях. При наличии контроллера и запущенного сервиса Leap Service, с помощью `AnyEvent::LeapMotion` можно получать данные о жестах и положении кончиков пальцев в пространстве.

- Perl::PrereqScanner::Lite

`Perl::PrereqScanner::Lite` — это новый сканер для поиска зависимостей модулей. Для лексического разбора используется быстрый `Compiler::Lexer`, что даёт высокую скорость обработки.

- File::Sip

`File::Sip` предназначен для чтения больших файлов, размер которых превышает доступную память. Модуль по понятным причинам уступает в скорости таким модулям, как `File::Slurp::Tiny`, но при этом имеет значительно меньшее потребление памяти (обычно 1/20 размера файла).

- Web::ChromeLogger

С помощью модуля `Web::ChromeLogger` можно производить отладку вашего веб-приложения в консоли браузера Chrome. Специальный плагин браузера `Chrome Logger` анализирует HTTP-заголовок `X-ChromeLogger-Data`, в котором помещаются отладочные сообщения, которые были сгенерированы во время работы приложения, и визуализирует полученные данные в консоли браузера.

Обновлённые модули

- Search::Elasticsearch 1.10

Модуль `Elasticsearch` вновь был переименован, на этот раз в `Search::Elasticsearch`. Связано это прежде всего с тем, что после предыдущего переиме-

нования из `ElasticSearch` пользователи систем с регистронезависимой файловой системой не могли установить обе версии модуля одновременно.

- Term::ReadLine::Gnu 1.24

Реализация `Term::ReadLine` на основе GNU библиотеки Readline была обновлена после почти четырёхлетнего перерыва. В новой версии добавлена поддержка новых функций и переменных библиотеки Readline версий 6.1 и 6.3, исправлены некоторые ошибки.

- DateTime 1.08

В новом релизе `DateTime` для вычисления текущего времени использует приватную

функцию `_core_time()` вместо непосредственного вызова встроенной функции `time()`. Это позволит упростить создание тестов для программ, в которых требуется подменять значение текущего времени. Для этого теперь требуется локально переопределить функцию `DateTime::_core_time()`. Одним словом, `DateTime` остаётся верен традициям нелогичного и дефектного API.

- Log::Log4perl 1.43

В новой версии `Log::Log4perl` добавлен формат для сообщений `%m{indent}`, который дополняет начальными отступами многострочные сообщения.

- ShardedKV 0.19

Новый релиз абстрактного интерфейса к распределённому хранилищу ключей-значений `ShardedKV` содержит несколько малозначительных исправлений. Официально заявлено, что модуль больше не считается экспериментальным и пригоден к использованию в рабочем окружении.

- Mouse 2.1.1

В новой версии модуля `Mouse` исправлена ошибка сборки модуля на системах с Perl 5.8.8 (RHEL 5).

- Perl::Tidy 20140328

Релиз 20140328 модуля для форматирования исходного кода `Perl::Tidy`

исправляет несколько ошибок, включая CVE-2014-2277 — небезопасная работа с временными файлами.

- Wx 0.9923

Интерфейс к кроссплатформенной GUI-библиотеке wxWidgets теперь поддерживает новую мажорную версию библиотеки 3.0.0.

■ *Владимир Леттиев*

7 Интервью с Екатериной Трефиловой

Екатерина Трефилова — Perl-программист, участник Perl-мероприятий, любитель мопсов и шоколада.

Когда и как начала программировать?

Когда мне было 11 лет, отец принес домой компьютер. В нем у меня было установлено несколько игрушек и Photoshop. Но интереснее всего было смотреть, что же внутри самой машины. С компьютером шла книга, рассказывающая, что и как работает. Одна из немногих инструкций к технике, которую я внимательно прочитала :) Это, конечно, скорее знакомство с компьютерами, а программировать я начала, когда мне было лет 13. Мне в руки попала книга с задачками по Pascal. Всякие звездочки

и квадраты. Меня восхитило, что умение правильно описывать задание позволяет сделать так много. С этого момента я влюбилась в программирование. К тому же я всегда очень любила рисовать. Знакомство с Photoshop продолжилось интересом к web-дизайну, который в свою очередь перерос в интерес к web-разработке.

Какой редактор используешь?

VIM, он великолепен! Я долгое время использовала Sublime, пробовала Komodo и Geany. Но всегда хотелось что-то подкрутить. Потом я познакомилась с VIM. Почитала про команды и настройку. Теперь даже не представляю как пользоваться чем-то еще, всегда хочется закончить работу, набрав `:w`.

Когда и как познакомилась с Perl?

На своей первой работе. Я училась на 4-м курсе. В университете проекты мы писали в основном на Python, а на работе требовалось поддерживать старый и разрабатывать новый функционал на Perl. Сначала было тяжело. Опыта не было, все вокруг о Perl ничего не знали. Поэтому не сразу наткнулась на книгу «Изучаем Perl», о чем очень жалею. Сильно сэкономила бы время. В интернете познакомилась с Ярославом Коршаком, он мне очень помог тогда. Рассказал, что нужно обязательно прочитать, что желательно. Когда разобралась немного, влюбилась в Perl. Можно делать все и разными способами. Эта невероятная свобода меня покорила!

С какими другими языками интересно работать?

Последнее время мне очень интересны

функциональные языки программирования: Lisp, Haskell. Прекрасное чувство, когда мозг трещит, перестраиваясь. Но писать на них что-то серьезное нет возможности. Очень часто работаю с JavaScript. Иногда играюсь с Perl6, чтобы знать, что с ним происходит, что нового в нем появилось.

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

Его свобода! Наверное, это можно считать и недостатком, но для меня эластичность Perl именно достоинство. Он позволяет решить любую задачу разными способами. Можно не просто сделать все, можно сделать все так, как хочется тебе. И конечно, одно из главных преимуществ Perl, - люди, которые на нем пишут. У Perl прекрасное, дружное сообщество, я не видела такого ни у одно-

го из других языков. Люди просто живут этим, наслаждаются своей работой, делают что-то новое и делятся этим с друг другом.

Какими, по-твоему, свойствами должны обладать языки будущего?

Думаю, в выигрышном положении находятся языки, исполняемые в браузере. Все жду, когда в этой сфере кто-нибудь начнет конкурировать с JavaScript. Если предположить, что такой конкурент появился, у него должен быть интуитивно понятный синтаксис. Если оглянуться вокруг, то можно заметить, что все больше людей интересуется программированием. Им должно быть удобно и приятно писать код.

Как, по-твоему, можно объяснить то, что большинство программистов это мужчины?

Наверное, дело в стереотипах.

Я училась на мех-мате, и в моей группе девушек было больше, чем парней. Математика и программирование даются женщинам ничуть не тяжелее, а порой даже легче, ведь мы усидчивые и больше приспособлены к кропотливой работе. Но по окончанию университета практически все девушки выбрали какие-то другие профессии. Сказалось влияние общественного мнения о том, что программист это мужская работа.

Чувствовала ли ты когда-нибудь дискриминацию или предвзятое отношений от коллег-мужчин?

К сожалению, да. С такими мужчинами очень сложно работать. Когда я столкнулась с таким отношением первый раз,

очень переживала, пыталась доказать, что меня недооценивают. И злилась, очень злилась. А потом поняла, что меня это совершенно не касается. Ведь от того, что кто-то не верит в мои способности, я не пишу код хуже. Просто перестала обращать внимание, и жить стало легче.

Последнее время все больше и больше скандалов в программисткой среде связаны с половой принадлежностью. Проблема действительно существует или она надумана? Отличается ли отношение в разных языковых сообществах?

Честно говоря, не слышала о подобных скандалах. Большинство программистов умные, вежливые люди, не склонные к шовинизму. Многие, наоборот, рады девушкам в сообществе и с готовностью помогают освоится. Не стоит судить по

нескольким исключениям. Отличается ли отношение в разных языковых сообществах? Чем меньше девушек пишет на этом языке программирования, тем больше мужчин из его сообщества скептически к ним относится. И все равно, таких мужчин будет очень мало. Надо помнить, что еще больше мужчин рады девушкам в своих рядах.

Нужно ли в особенном порядке фиксировать нормы поведения (Code of conduct, CoC) на конференциях?

Каверзный вопрос. Ты имеешь в виду какие-то правила, фиксирующие общение между мужчинами и женщинами? Думаю, такие правила не нужны, мужчины и женщины равны и правила могут быть общие. Если речь о соблюдении общих норм поведения, то такие правила, конечно, должны

быть. Ведь это общественное мероприятие, и на нем всем должно быть комфортно.

Где работаешь сейчас? Какие задачи приходится решать с помощью Perl?

Сейчас я работаю программистом в Mail.Ru Group, одной из крупнейших интернет-компаний в русскоязычном интернете, в проекте Афиша Mail.Ru. Бэкенд проекта полностью написан на Perl, так что все задачи, которые я решаю, я решаю на Perl. Занимаюсь поддержкой существующего функционала и разработкой нового. Мне очень повезло с командой, талантливые и увлеченные люди. С ними интересно и приятно работать. Тут очень здорово!

Как приобщать девушек к программированию? Стоит ли советовать учить Perl?

Мне кажется, нужно прекратить считать, что девушки-программисты что-то особенное. И что «не женское это дело». Выбирая профессию, многие пугаются такого отношения. Стоит ли советовать учить Perl? Конечно стоит! Это прекрасный язык, позволяющий сделать так много интересного!

Вопросы от читателей

Нравится ли посещать Perl-конференции?

Я обожаю Perl-конференции. Моя первая конференция - YAPC::Russia + Perl Mova 2012. Там я увидела, сколько интересных вещей пишут на Perl и познакомилась с интересными людьми. После этой конференции так зарядилась энергией, что переехала в Москву!

Как тебе опыт выступления с докладом?

Мне очень понравилось, планирую повторить :) Выступать очень полезно, пока ты готовишься, можно иначе взглянуть на тему. Во время выступления осознаешь, что есть люди, которым эта тема тоже интересна, у них могут быть похожие проблемы и совсем другие решения. Это все очень мотивирует.

■ *Вячеслав Тихановский*