PRAGMATIC PERL

27

05/2015

pragmaticperl.com

Pragmatic Perl 27

pragmaticperl.com

Выпуск 27. Май 2015

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

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

Авторы статей: Андрей Шитов, Владимир Леттиев

Обложка: Марко Иванык

Корректоры: Георгий Бажуков, Андрей Шитов

Выпускающий редактор: Вячеслав Тихановский

Ревизия: 2015-05-06 15:02

Оглавление

1. От редактора

Напоминаем, что 16 и 17 мая в Москве пройдет конференция YAPC::Russia 2015!

Также напоминаем, что у нас есть форум на котором можно обсудить различные Perl-темы и не только.

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. Отладка приложений на AnyEvent

Отладка асинхронных приложений часто затруднена из-за нелинейного характера выполняемого кода, когда ошибки проявляются под нагрузкой или в каких-то исключительных трудновоспроизводимых ситуациях. Привычные инструменты могут быть неудобны и неинформативны. Если приложение построено на основе AnyEvent, то для него уже существуют готовые рецепты отладки.

AnyEvent::Log

> Самое эффективное средство отладки — по-прежнему тщательное размышление с правильно расставленными операторами print.
>
> – Брайан Керниган, «Unix for Beginners» (1979)

AnyEvent::Log — это фреймворк для ведения журнала событий. Основное его достоинство в том, что он поставляется вместе с дистрибутивом AnyEvent и может лениво загружаться, как только вы первый раз вызываете метод `AE::log` в своём коде. Кроме того, сам AnyEvent также начинает вести журналирование своих внутренних событий, позволяя глубже понимать происходящее. По сути, `AnyEvent::Log` просто даёт публичный доступ к внутренним механизмам ведения логов, применяющимся внутри AnyEvent.

Концепция журналирования `AnyEvent::Log` довольно запутанная, но есть много общего с привычными `Log::Dispatch` и `Log::Log4perl`, например, контекст, уровни детализации и контейнеры для вывода сообщений. Начать успешно использовать `AnyEvent::Log` можно опустив практически все тонкости его организации.

```
1 use AnyEvent;
2
3 AE::log trace => "прочитан фрагмент статьи в 200 байт";
4 AE::log debug => "функция total_readed вернула 500";
5 AE::log info  => "читаю статью";
6 AE::log note  => "функция understood_readed вернула 0, пробую еще
      раз";
7 AE::log warn  => "не понимаю, что тут написано"
8 AE::log error => "строкой выше пропущена точка с запятой";
9 AE::log critical => "место для записи закончилось, сохраняю в
    памяти";
```

```
10 AE::log alert => "больше свободной памяти нет";
11 AE::log fatal => "в статье не найдены котики, продолжать чтение
     невозможно";
```

В данном примере сообщения разделяются по уровню важности или детализации действий программы, каждый уровень имеет своё цифровое значение — чем выше значение, тем более высокий уровень детализации:

- 9 — **trace** — трассировочные сообщения, подразумевающие обильный вывод для каждого шага программы;
- 8 — **debug** — отладочные сообщения, применяемые, как правило, для подробного вывода входных параметров функций или её результатов;
- 7 — **info** — информационные сообщения о нормальном течении программы, например, для веб-сервера это сообщения о доступе к ресурсу в access.log;
- 6 — **note** — сообщение о необычном или граничном условии;
- 5 — **warn** — предупреждение, не обязательно ошибка, но, вполне вероятно, проблема;
- 4 — **error** — некритичная ошибка в программе;
- 3 — **critical** — критическая ошибка, отказ резервирования;
- 2 — **alert** — критическая ошибка, отказ основной системы;
- 1 — **fatal** — фатальная ошибка, дальнейшая работа программы невозможна.

По умолчанию AnyEvent устанавливает уровень детализации error, это означает, что выводиться будут сообщения с уровнями с 1 по 4 (alert — error), а все сообщения с большим уровнем подробности отсекаются.

Регулировать уровень детализации можно с помощью переменной окружения `PERL_ANYEVENT_VERBOSE` (или более короткий алиас `AE_VERBOSE`), например, задать самый подробный уровень trace:

```
1 $ export PERL_ANYEVENT_VERBOSE=9
2 $ ./foo.pl
```

По умолчанию весь вывод идёт на `STDERR`. Это можно изменить через переменную окружения `PERL_ANYEVENT_LOG` (или `AE_LOG`), например, задав имя файла, куда направлять весь вывод:

```
1 $ export PERL_ANYEVENT_LOG=log=file=/some/path
```

Или, например, направить весь вывод в syslog:

```
$ export PERL_ANYEVENT_LOG=log=syslog
```

Иерархия контекстов AnyEvent::Log

Мы рассмотрели самый простой способ использования `AnyEvent::Log`, в котором уровень детализации и направление вывода задаётся глобально для всего приложения. В некоторых случаях требуется более тонкая и детальная настройка. `AnyEvent::Log` предоставляет подобную возможность, но, чтобы разобрать дальнейшие рецепты, требуется понимание *контекста журналирования* и *иерархии контекстов*.

Каждое сообщение ассоциируется со своим контекстом журналирования. `AnyEvent::Log` создаёт контекст на основе имени пакета вызывающего кода. Например,

```
package Foo::Bar;
use AnyEvent;

sub baz {
    AE::log error => "вызов из baz пакета Foo::Bar"
}

package main;
use AnyEvent;

AE::log error => "вызов из main";
Foo::Bar::baz
```

Получим следующий вывод:

```
2015-04-27 19:29:30.000000 +0300 error main: вызов из main
2015-04-27 19:29:30.000000 +0300 error Foo::Bar: вызов из baz
    пакета Foo::Bar
```

Каждое сообщение имеет метку времени и значение уровня отладки. Непосредственно само сообщение предваряет заголовок текущего контекста, которым в данном примере является имя пакета. Таким образом, первое сообщение из основного пакета `main` выполняется из контекста `main`, а вызов `log` из функции `baz` находится в контексте `Foo::Bar`.

Каждый контекст журналирования выполняет три основные функции:

1. *Фильтрация.* Каждый контекст устанавливает уровень детализации или, иначе, маску журнала. Сообщения, которые не попадают в диапазон уровней, игнорируются (маскируются).
2. *Запись.* Для записи сообщений каждый контекст создаёт две функции:
    1. функцию форматирования, которой передаётся временная метка, контекст, уровень и само сообщение. Данная функция отвечает непосредственно за форматирование строки сообщения;
    2. функцию записи, которая отвечает за запись сообщения и его дальнейшее распространение.
3. *Распространение.* Каждый контекст может иметь несколько подчинённых контекстов. Если сообщение не было отфильтровано на этапе фильтрации и не было «поглощено» на шаге записи, то в этом случае контекст передаёт сообщение всем подчинённым контекстам.

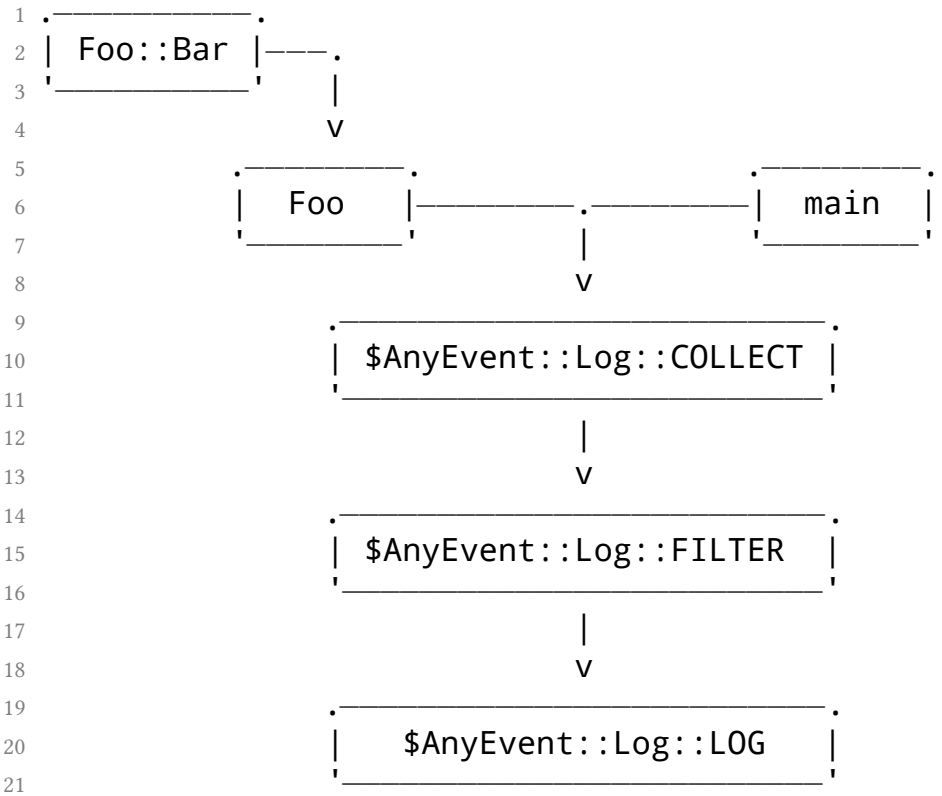
По умолчанию все контексты имеют полный набор уровней детализации (т.е. ничего не фильтруют), но у них отключены функции форматирования и записи, поэтому по умолчанию все контексты просто передают сообщения дальше подчинённым контекстам.

Каждому контексту привязывается ровно один подчинённый контекст — это контекст, который имеет имя пакета текущего контекста за исключением последнего компонента. Например, в указанном выше примере создаётся контекст `Foo::Bar`, к которому подключается подчинённый контекст `Foo`. Если же контекст имеет только один компонент в имени пакета, то по умолчанию подчинённым контекстом становится специальный контекст `$AnyEvent::Log::COLLECT`, который создаёт `AnyEvent::Log`.

Всего `AnyEvent::Log` создаёт три специальных контекста:

1. `$AnyEvent::Log::COLLECT` — это глобальный контекст, к которому сходится вся иерархия контекстов пакетов.
2. `$AnyEvent::Log::FILTER` — контекст, который становится подчинённым контекста `$AnyEvent::Log::COLLECT` и его основная задача — фильтровать все сообщения, уровень которых выше, чем установленный в переменной `PERL_ANYEVENT_VERBOSE`.
3. `$AnyEvent::Log::LOG` — финальный контекст, который становится подчинённым `$AnyEvent::Log::FILTER` и выводит все сообщения с помощью функции `warn`.

Если взять описанный выше пример, то получаем такую иерархию контекстов:



Помимо контекстов журналирования, входящих в подобную иерархию, можно создавать *анонимные контексты*, которые не имеют подчинённых контекстов. Такие объекты удаляются, как только на них перестают ссылаться.

Все неанонимные контексты хранятся в публичном хеше `%AnyEvent::Log::CTX`, где в качестве ключа используется имя пакета. При желании их всегда можно проинспектировать.

Для чего нужна подобная иерархия? Она позволяет гибко настраивать поведение каждого контекста, что будет наглядно продемонстрировано далее.

Глобальный уровень детализации Чтобы установить глобальный уровень детализации для всех сообщений, достаточно задать уровень детализации контекста `$AnyEvent::Log::FILTER`:

```
# Установить глобальный уровень детализации в info
$AnyEvent::Log::FILTER->level ("info");
```

Разумеется, то же самое можно было сделать в контексте COLLECT, так как через него проходят все сообщения, но делать этого не рекомендуется, поскольку задача COLLECT — сбор сообщений, которые потом можно распределить на другие контексты.

Запись сообщений в файл вместо вывода на STDERR Как уже было сказано, функция-колбек для записи сообщений по умолчанию определена только в $AnyEvent::Log::LOG, который выводит их на STDERR. Таким образом, чтобы переопределить эту запись, нужно задать запись в файл, например так:

```
# Специальный метод для установки записи в файл
$AnyEvent::Log::LOG->log_to_file ($path);
```

Опять же, то же самое можно было бы задать на уровне $AnyEvent::Log::FILTER, но это некорректно, поскольку задача контекста фильтрации в другом. Кроме того, в этом случае сообщения по-прежнему бы выводились на STDERR, так как метод записи в $AnyEvent::Log::LOG не был бы переопределён.

Различные уровни детализации: глобальный и заданного пакета Добавим к нашему первоначальному примеру следующий фрагмент:

```
# Получить контекст пакета 'Foo'
my $foo_ctx = AnyEvent::Log::ctx('Foo');

# Задать уровень info
$foo_ctx->level('info');

# Функция логирования
$foo_ctx->log_cb( sub {
    warn shift;

    # 0 - разрешает дальнейшее распространение
    #     сообщения по иерархии
    # 1 - запрещает дальнейшее распространение
    #     сообщения по иерархии
    0;
});
```

Теперь журналирование для всех пакетов под иерархией пакета ‘Foo’ будет вестись на уровне info, в то время как глобальный уровень останется таким,

как задан в переменной окружения AE_VERBOSE. Нужно обязательно задать функцию-колбек для записи, так как по умолчанию у всех контекстов эта функция не установлена.

Дублирование всех сообщений в файл Предположим, что требуется дополнительно сохранять все поступающие сообщения с любым уровнем детализации в файл независимо от глобального фильтра уровня детализации. Поскольку требуется получать все сообщения иерархии, то удобно создать новый анонимный контекст и сделать его подчинённым для $AnyEvent::Log::COLLECT: этот контекст получает все сообщения от иерархии, и они ещё не были отфильтрованы глобальным фильтром в контексте $AnyEvent::Log::FILTER.

```
1 # анонимный контекст
2 my $annon_ctx = AnyEvent::Log::Ctx->new( log_to_file => $path );
3
4 # подключаем его как подчинённый контекст к $AnyEvent::Log::
    COLLECT
5 $AnyEvent::Log::COLLECT->attach( $annon_ctx );
```

Оптимизации

После того как стала понятна концепция журналирования в AnyEvent::Log, можно перейти к некоторым полезным фокусам с оптимизацией. Например, оптимизация по скорости выполнения.

Как правило, сообщения уровней *trace* и *debug* могут вызываться очень часто и могут ощутимо замедлять работу приложения, когда оно работает в нормальном режиме без глубокого уровня детализации, из-за накладных расходов на вызов функции. Для этого случая предусмотрена следующая оптимизация:

```
1 use AnyEvent::Log;
2
3 # Создаём трассировщик, переменная $trace
4 # хранит его состояние (активен/неактивен)
5 my $tracer = AnyEvent::Log::logger trace => \my $trace;
6
7 # Если нет режима trace, то $trace — это ложь
8 # и функция не вызывается
9 $tracer->("trace message") if $trace;
```

```
10
11 # или так
12 $trace and $tracer->("trace message");
```

Тут важно отметить, что переменная `$trace` передаётся как ссылка. Таким образом, её значение может устанавливаться и обновляться в случае, если уровень детализации меняется по ходу выполнения программы.

Также можно использовать ленивое создание сообщений:

```
1 use AnyEvent::Log;
2 use Data::Dumper;
3
4 AE::log debug => sub {
5     Dumper $some_big_structure
6 };
```

В данном примере функция `AE::log` будет вызвана при любом уровне детализации, но функция, возвращающая сообщение, вызывающая `Dumper` большого объекта, будет вызываться только в режиме debug, что существенно ускоряет работу при низком уровне детализации.

AnyEvent::Strict

Модуль `AnyEvent::Strict` при загрузке включает строгую проверку параметров, передаваемую функциям AnyEvent, что несколько замедляет работу, но позволяет выявить типичные ошибки. Кроме того все функции-колбеки оборачиваются в специальную функцию, которая проверяет, что вызываемый колбек не модифицирует переменную `$_`, что явно свидетельствует об ошибке.

Как правило, включать `AnyEvent::Strict` в коде не обязательно, достаточно установить переменную окружения `PERL_ANYEVENT_STRICT` при запуске AnyEvent-приложения. Это позволит быстрее отловить некоторые простые ошибки и избавит от рутинной отладки.

AnyEvent::Debug

В случае, если журнал приложения и строгий режим проверки параметров не дают результатов при поиске ошибок, то со следующим этапом отладки может помочь `AnyEvent::Debug`.

Основа `AnyEvent::Debug` — это интерактивная командная строка, встраиваемая в приложение. После запуска приложения вы в любой момент можете буквально «зателнетиться» в работающее приложение и получить возможность запускать различные команды, которые будут выполняться в контексте запущенного кода. В простейшем случае можно проинспектировать значения глобальных переменных.

Итак, в начало отлаживаемой программы требуется добавить следующие строки:

```
1 use AnyEvent::Debug;
2
3 my $shell = AnyEvent::Debug::shell "unix/", "/path/to/sock";
```

Функция `AnyEvent::Debug::shell` в данном случае создаёт файловый сокет. Вместо файлового сокета можно указать в параметрах IP-адрес и порт:

```
1 my $shell = AnyEvent::Debug::shell '127.0.0.1', 2222;
```

Но, поскольку никакой авторизации для доступа в командную строку нет, то это может быть небезопасно, и в общем случае рекомендуется использовать файловые сокеты, доступ к которым можно легко ограничивать при помощи стандартных прав файлового доступа.

После запуска приложения создаётся сокет, к которому можно подключиться с помощью стандартных утилит. В случае файлового сокета можно использовать утилиту *socat*. Например:

```
1 $ socat readline,history=.ae–debug–history unix:/path/to/sock
```

В данном примере socat откроет сокет `/path/to/sock` и при этом загрузит библиотеку GNU readline для работы со стандартным вводом. Параметр *history* позволяет указать файл, в котором будет сохраняться история команд, набираемых в командной строке, что может быть очень удобно при частом подключении.

Команды оболочки AnyEvent::Debug

Интерактивная строка является обычным REPL (Read-Eval-Print Loop), позволяя выполнять Perl-код в контексте приложения. Кроме того, доступно несколько встроенных команд, которые позволяют работать с обработчиками событий.

`help` Справка по доступным командам.

`v [level]` При подключении командной оболочки, создаётся контекст журналирования `$LOGGER`, который подключается к контексту `$AnyEvent::Log::COLLECT` приложения, что позволяет захватывать все отладочные сообщения приложения и выводить их прямо в консоли.

Команда `v` позволяет регулировать уровень детализации вывода логов контекста `$LOGGER` в диапазоне от 0 до 9.

`wr [level]` Любое приложение, работающие на AnyEvent создаёт обработчики событий. Команда `wr` позволяет включать/выключать режим добавления специальной обёртки для вновь добавляемых обработчиков событий. Возможны три значения режима:

- 0 — Отключает режим обёртки.
- 1 — Включает режим обёртки. Теперь, когда в приложении происходит создание нового обработчика, создаётся объект класса `AnyEvent::Debug::Wrapped`, который сохраняет информацию о месте в программе, где был создан обработчик, оборачивается функция-колбек обработчика, добавляя трассировочный вывод. Также добавляется счётчик, который считает количество вызовов функции-колбека.
- 2 — То же самое, что и уровень 1, только дополнительно сохраняет полный бектрейс в точке создания обработчика в коде. Это существенно замедляет создание обработчиков, но даёт детальную информацию о стеке вызовов.

Кроме того, включить режим создания обёртки можно и при запуске приложения, установив переменную окружения PERL_ANYEVENT_DEBUG_WRAP.

`wl 'regex'` Команда `wl` позволяет вывести список обёрток обработчиков. Если регулярное выражение опущено, то выводится весь список.

`i id` Команда `i` позволяет вывести детальную информацию об обработчике по его уникальному номеру.

`t/u [id]` Команда `t` — включает режим трассировки всех вновь создаваемых обработчиков (по умолчанию), а команда `u` — отключает режим трассировки. Также, если указан номер, то можно включать/выключать трассировку индивидуально для каждого обработчика.

`coro [command]` Команда позволяет переключиться в оболочку `Coro::Debug`, если она доступна, и выполнить её команды. Полезно, если приложение также использует модуль `Coro`.

Пример отладки

Рассмотрим пример отладки веб-сервера Twiggy, который является приложением, работающим на AnyEvent.

С помощью утилиты `plackup` мы запускаем тривиальное веб-приложение и создаём файловый сокет `twiggy.sock` для отладочного подключения.

```
$ PERL_ANYEVENT_DEBUG_WRAP=2 \
    plackup -s Twiggy \
      -MAnyEvent::Debug \
      -e 'AnyEvent::Debug::shell "unix/", "twiggy.sock";
          sub { [ 200, [], ["hello, world"] ] }'
```

Переменная окружения PERL_ANYEVENT_DEBUG_WRAP включит режим обёртки с самого запуска программы, чтобы отловить создание всех обработчиков.

Подключимся к файловому сокету и получим приглашение:

```
$ sockat readline unix:twiggy.sock
Welcome, unix/:, use 'help' for more info!
>
```

Теперь включим самый высокий уровень детализации:

```
> v 9

verbose logging is now enabled.
```

Получим список всех обработчиков:

```
> wl

12261272  (eval 16):1(Plack::Runner::ANON)>io>AnyEvent::Socket
   :489
12864152  AnyEvent::Debug:527((eval))>io>AnyEvent::Debug:112
19500264  AnyEvent::Debug:539(Wrap::ANON)>io>AnyEvent::Base::
   _signal_exec
19403480  Twiggy::Server:598(run)>signal>Twiggy::Server:27
19326992  Twiggy::Server:79(_create_tcp_server)>io>AnyEvent::
   Socket:489
```

Каждый обработчик указан в отдельной строке. Первой колонкой записан адрес объекта-обёртки, далее указан номер строки, где был создан обработчик и какого он типа (io, signal).

Первые два обработчика относятся непосредственно к `AnyEvent::Debug`: сам unix-сокет, сокет текущего подключения (socat). Далее идёт служебный пайп модуля `Async::Interrupt`, который асинхронно отлавливает сигналы для AnyEvent.

Последние два обработчика были созданы непосредственно Twiggy: обработчик сигнала `QUIT` и непосредственно прослушиваемый TCP-сокет для подключения веб-клиентов.

Можно посмотреть последний более детально с помощью команды `i`:

```
> i 19326992

19326992 Twiggy::Server:79(_create_tcp_server)>io>AnyEvent::
   Socket:489
type:    io watcher
args:    fh GLOB(0x126e258) poll r
```

```
created: 2015-04-28 14:01:19.084551 +0300 (1430218879.08455)
file:    .../lib/site_perl/5.20.1/Twiggy/Server.pm
line:    79
subname: Twiggy::Server::_create_tcp_server
context:
tracing: enabled
cb:      CODE(0xbeeff0) (AnyEvent::Socket:489)
invoked: 0 times
...
```

В выводе для краткости опущен длинный вывод трассировки стека до точки создания обработчика. Каждая строка описывает все доступные характеристики обработчика, это и тип, аргументы, время создания, файл исходного кода, строка и подпрограмма. Также видно и количество вызовов функции-колбека.

Попробуем выполнить тестовый запрос к серверу:

```
$ curl http://0:5000/
```

В отладочной консоли можно будет увидеть следующий вывод (метки времени опущены для краткости) :

```
debug AnyEvent::Base: using Time::HiRes for sub-second timing
    accuracy.
trace AnyEvent::Debug: enter Twiggy::Server:79(_create_tcp_server
    )>io>AnyEvent::Socket:489
trace AnyEvent::Debug: leave Twiggy::Server:79(_create_tcp_server
    )>io>AnyEvent::Socket:489
```

Первое сообщение относится к AnyEvent, который загрузил `Time::HiRes` для получения временных меток большей точности. Следующее сообщение оставляет обёртка обработчика tcp-сокета Twiggy, видно что обработчик запустился *(enter)* и завершился *(leave)*. Указано точное место обработчика в исходном коде.

Если теперь повторно запросить информацию по обработчику, то можно увидеть, что счётчик вызовов увеличился на 1.

```
> i 19326992
...
invoked: 1 times
...
```

Теперь попробуем подключиться с помощью telnet к веб-серверу Twiggy.

```
$ telnet 0 5000
```

В консоли появилось больше сообщений:

```
14:10:28.747 trace AnyEvent::Debug: enter Twiggy::Server:79(
    _create_tcp_server)>io>AnyEvent::Socket:489
14:10:28.748 trace AnyEvent::Debug: creat Twiggy::Server:209(
    _create_req_parsing_watcher)>timer>Twiggy::Server:27
14:10:28.749 trace AnyEvent::Debug: creat Twiggy::Server:224(
    _create_req_parsing_watcher)>io>Twiggy::Server:27
14:10:28.749 trace AnyEvent::Debug: leave Twiggy::Server:79(
    _create_tcp_server)>io>AnyEvent::Socket:489
```

Первое и последние сообщения относятся к обработчику TCP-сокета: функция-колбек приняла подключение. В процессе работы обработчика он создаёт два новых обработчика с типами *timer* и *io* в функции `_create_req_parsing_watcher`. Если заглянуть в исходный код Twiggy, то видно, что функция `_create_req_parsing_watcher` вызывается в случае, если подключение есть, но данных в нём ещё нет. Создаваемый io-обработчик ждёт данных в сокете от клиента. А timer-обработчик создаётся для того, чтобы реализовать функционал таймаута. В случае с запуском curl этих обработчиков не было, поскольку curl без промедления отправил все данные запроса в сокет.

Команда `wl` покажет нам эти два новых обработчика:

```
> wl '_create_req'

19477640  Twiggy::Server:209(_create_req_parsing_watcher)>timer>
    Twiggy::Server:27
19568344  Twiggy::Server:224(_create_req_parsing_watcher)>io>
    Twiggy::Server:27
```

Посмотрим обработчик таймера:

```
> i 19477640

...
args:    after 300 interval 0
...
```

Как видно, таймаут составляет 300 секунд, и, если мы подождём эти 5 минут, то увидим следующие сообщения:

```
14:15:28.748 trace AnyEvent::Debug: enter Twiggy::Server:209(
   _create_req_parsing_watcher)>timer>Twiggy::Server:27
14:15:28.748 trace AnyEvent::Debug: dstry Twiggy::Server:224(
   _create_req_parsing_watcher)>io>Twiggy::Server:27
14:15:28.749 trace AnyEvent::Debug: leave Twiggy::Server:209(
   _create_req_parsing_watcher)>timer>Twiggy::Server:27
14:15:28.749 trace AnyEvent::Debug: dstry Twiggy::Server:209(
   _create_req_parsing_watcher)>timer>Twiggy::Server:27
```

Видно, что запускается обработчик таймера, который уничтожает io-обработчик, то есть закрывает соединение с клиентом. После завершения функции-колбека таймера сам обработчик таймера также уничтожается.

В данном примере мы рассмотрели способ изучения работы асинхронного приложения, наблюдая за его обработчиками.

Доступ к переменным приложения

Иногда полезно получить доступ к переменным и структурам данных приложения. На этот счёт в документации `AnyEvent::Debug` есть простая рекомендация: использовать глобальные переменные. Например, сам `AnyEvent::Debug` хранит все обёртки к обработчикам в хеше `%AnyEvent::Debug::Wrapped`, которые легко проинспектировать в отладочной консоли:

```
> use DDP

> p %AnyEvent::Debug::Wrapped

{
 12261272   AnyEvent::Debug::Wrapped  {
  public methods (4) : DESTROY, id, trace, verbose
  private methods (1) : ANON
  internals: {
   arg      {
    fh     *AnyEvent::Socket::$state{...}  (read/write, flags:
       nonblocking, layers: unix perlio),
    poll   "r"
   },
...
```

Также можно использовать классы-синглтоны, которые позволяют через вызов метода получать какие-либо данные.

В крайнем случае можно воспользоваться модулем PadWalker, который, используя чёрную магию XS, извлекает на свет лексические переменные. Например,

```
> i 19403480
...
cb:      CODE(0xbc1f38) (Twiggy::Server:27)
...
```

Как видно, нам доступна ссылка на код обработчика. С помощью функции PadWalker::peek_sub мы можем проинспектировать лексические переменные, определённые в этой функции.

```
> use DDP

> use PadWalker

> p PadWalker::peek_sub $AnyEvent::Debug::Wrapped{19403480}{cb}
{
 $self   Twiggy::Server {
 ...
 $w AnyEvent::Debug::Wrapped {
 ...
}
```

Как видно, в выводе есть объект $self класса Twiggy::Server, который можно проинспектировать.

Мониторинг приложений

Помимо отладки приложения, можно использовать шелл-сокет AnyEvent::Debug для мониторинга состояния обработчиков. Например, периодическое подключение со считыванием числа активных обработчиков или счётчиков вызовов определённых функций-колбеков. Это становится возможным даже если в самом приложении не было предусмотрено никакого API для мониторинга.

Заключение

Дистрибутив `AnyEvent` содержит полезные инструменты для отладки и инспектирования работы приложений такие как: `AnyEvent::Log`, `AnyEvent::Strict` и `AnyEvent::Debug`. Их использование позволяет отлаживать код «на горячую» в работающем приложении, не требуя перезапуска и минимизируя побочные эффекты вмешательства в нормальный ход работы программы.

■ *Владимир Леттиев*

3. Операторы Perl 6. Часть 1

Обзор префиксных, постфиксных и инфиксных операторов Perl 6

Многие операторы, доступные в Perl 6, понятны без объяснения даже тем, кто не знаком с Perl 5. В этой справочной статье перечислены все операторы, и там, где это необходимо, сделаны пояснения с примерами.

Операторы можно разделить на несколько групп в зависимости от их синтаксических особенностей. Это префиксы, инфиксы, постфиксы и еще несколько групп, которые будет рассмотрены в следующей части.

Префиксы (prefixes)

!

`!` — логический оператор отрицания.

```
say !True;     # False
say !(1 == 2); # True
```

+

`+` — унарный плюс, преобразующий свой операнд к числовому контексту. Выполняемое действие аналогично вызову метода `Numeric`:

```
my Str $price = '4' ~ '2';
my Int $amount = +$price;
say $amount;               # 42
say $price.Numeric;        # 42
```

Один из важных случаев применения унарного плюса встречался в статье про грамматики: `+$/`. Эта конструкция преобразует в число объект типа `Match`, который содержит данные о совпавшей части грамматики или регекса.

-

– — унарный минус, меняющий знак у числа. Поскольку оператор неявно вызывает метод `Numeric`, он может попутно преобразовать контекст, как это делает унарный плюс.

```
my Str $price = '4' ~ '2';
say −$price; # −42
```

?

? — оператор, преобразующий контекст в логический, вызывая на объекте метод `Bool`:

```
say ?42; # True
```

~

~ — оператор преобразования к строковому типу.

```
my Str $a = ~42;
say $a.WHAT; # (Str)
```

Иногда строковый контекст может быть указан неявно, например, при интерполяции переменных внутри строк в двойных кавычках.

++

++ — префиксная форма оператора инкремента. Сначала выполняется инкремент, а потом возвращается новое значение.

```
my $x = 41;
say ++$x; # 42
```

Инкремент не обязан ограничиваться только числами:

```
my $a = 'a';
say ++$a; # b
```

Особый случай — имена файлов с порядковым номером внутри (текстовые строки, содержащие число, точку и расширение):

```
my $f = "file001.txt";

++$f;
say $f; # file002.txt

++$f;
say $f; # file003.txt
```

–

–– — префиксная форма оператора декремента. Как и префиксный ++, оператор сначала выполняет действие над операндом (декремент в данном случае), а затем возвращает результат.

```
my $x = 42;
say ––$x; # 41
```

С текстовыми значениями поведение аналогично оператору ++.

+^

+^ — побитовое отрицание с учетом двоичного дополнения.

```
my $x = 10;
my $y = +^$x;
say $y; # –11 (не –10)
```

Этот оператор может выступать и в качестве бинарного (см. дальше). Ср. также с ?^.

?^

?^ — логическое отрицание. Важно обратить внимание, что это не инверсия битов: вначале аргумент преобразуется к булевому значению, а затем делается отрицание.

```
my $x = 10;
my $y = ?^$x;
say $y;       # False
say $y.WHAT;  # (Bool)
```

^

^ — оператор, создающий диапазон (объект типа Range) от нуля до указанного значения (не включая его).

```
.print for ^5; # 01234
```

Этот пример аналогичен более явному:

```
.print for 0..4; # 01234
```

|

| разворачивает объекты, содержащие несколько значений (например, массивы, пары или парселы), в список. Этот оператор необходимо использовать, в частности, при вызове функции, принимающей список скаляров, когда исходные данные находятся в массиве:

```
sub sum($a, $b) {
    $a + $b
}

my @data = (10, 20);
say sum(|@data); # 30
```

Без оператора | компилятор сообщит об ошибке, поскольку функция ожидает два скаляра и не готова принять массив.

not

not преобразует операнд к логическому типу и делает отрицание. То же, что делает префиксный оператор !.

```
say not False; # True
```

so

so преобразует операнд к логическому типу и возвращает результат. Действие аналогично префиксу ?.

```
say so 42;   # True
say so True; # True
say so 0.0;  # False
```

temp

temp — делает переменную временной, значение которой восстановится при выходе за пределы текущей области видимости (как local в Perl 5).

```
my $x = 'x';
{
    temp $x = 'y';
    say $x;        # y
}
say $x;            # x
```

Ср. с префиксом let.

let

let — префиксный оператор, позволяющий восстановить значение переменной, если выход из текущей области видимости был выполнен с исключением.

```
my $var = 'a';
try {
    let $var = 'b';
    die;
}
say $var; # a
```

При наличии die пример напечатает исходное значение a, а если строку с die закомментировать, то выполненное в блоке присваивание сохранится, и на печати окажется b.

Ключевое слово `let` выглядит похоже на декларатоpы `my`, `our` и другие, но является при этом префиксным оператором. Ср. с префиксом `temp`.

Постфиксы (postfixes)

`++`

`++` — постфиксный вариант инкремента. Инкрементирует операнд, но, в отличие от префиксной формы, возвращает предыдущее значение.

```
my $x = 42;
say $x++;   # 42
say $x;     # 43
```

`--`

`--` — постфиксный декремент. Уменьшает значение на единицу и возвращает предыдущее значение.

И постфиксные, и префиксные операторы инкремента и декремента имеют «магическое» (magic) свойство правильно обрабатывать числа в именах файла:

```
my $filename = 'file01.txt';
for 1..10 {
    say $filename++;
}
```

Этот пример печатает имена файлов `file01.txt`, `file02.txt`, ..., `file10.txt`.

Постфиксы вызова методов (method postfixes)

В Perl 6 есть несколько синтаксических конструкций, начинающихся с точки, которые несколько похожи на постфиксные операторы. Все они относятся к вызову методов на объекте.

.

`.method` вызывает метод `method` на переменной. Это работает как с настоящими методами классов, определенных пользователем, так и с объектами всех встроенных типов. Кроме того, при попытке вызвать метод на объекте нативного типа (такого как `int` — машинное представление целого числа), будет создан объект одного из встроенных типов (в примере с `int` — `Int`), на котором и вызовется метод.

```
say "0.0".Numeric; # 0
say 42.Bool;       # True

class C {
    method m() {say "m()"}
}
my $c = C.new;
$c.m(); # m()
```

.=

`.=` является мутирующим вызовом метода на объекте. Действие `$x.=method` равнозначно записи `$x = $x.method`.

В следующем примере контейнер `$o`, первоначально содержащий объект класса `C`, после вызова `$o.=m()` замещается новым объектом, уже другого класса.

```
class D {
}

class C {
    method m() {
        return D.new;
    }
}

my $o = C.new;
say $o.WHAT;  # (C)

$o.=m();
say $o.WHAT;  # (D)
```

.^

`.^method` вызывает метод `method`, но не напрямую на текущем объекте, а на метаобъекте типа `HOW`. Следующие две записи эквивалентны:

```
my Int $i;
say $i.^methods();
say $i.HOW.methods($i);
```

Метаобъекты — отдельная тема, к которой есть смысл вернуться в следующий раз.

.?

`.?method` вызывает метод, если он определен. Если нет, возвращает `Nil`.

```
class C {
    method m() {'m'}
}

my $c = C.new();
say $c.?m();     # m
say $c.?n();     # Nil
```

.+

`.+method` делает попытку вызвать все методы с именем `method`, доступные для объекта. Такая ситуация может возникнуть при создании иерархий объектов.

```
class A {
    method m($x) {"A::m($x)"}
}
class B is A {
    method m($x) {"B::m($x)"}
}

my $o = B.new;
my @a = $o.+m(7);
say @a; # Печатается B::m(7) A::m(7)
```

В этом примере объект `$o` может обратиться к методу `m` либо из своего класса `B`, либо из родительского `A`. Конструкция `$o.+m(7)` выполняет вызовы обеих методов и помещает результаты в массив. Если метода с нужным именем не обнаружено, возникает исключение.

`.*`

`.*method` вызывает все методы с именем `method`, и возвращает парсел со списком результатов, либо пустой набор, если метод не определен. В остальном работает аналогично оператору `.+`.

Инфиксные операторы

Инфиксные операторы стоят в программе между операндами, и могут быть бинарными (в отличие от унарных префиксных или постфиксных) или тернарными (такой оператор всего один).

Простейший пример инфиксного оператора — символ сложения `+`. Справа и слева он ожидает два значения, например, две переменные: `$a + $b`. Важно понимать, что один и тот же символ или одна и та же последовательность символов в разных контекстах может быть или префиксом, или инфиксом. В примере с плюсом — это определенный в Perl 6 унарный плюс, устанавливающий числовой контекст: `+$str`.

Операторы для работы с числами

`+`, `-`, `*`, `/`

`+`, –, `*`, `/` — операторы, выполняющие соответствующие арифметические действия, они не требуют пояснения. При разговоре о Perl 6 нужно помнить, что прежде чем выполнить операцию операнды будут автоматически приведены к численному (`Numeric`) типу, если это необходимо.

div

`div` — целочисленное деление с округлением вниз.

```
say 10 div 3;  # 3
say −10 div 3; # 4
```

%

`%` — деление по модулю (остаток от целочисленного деления). Операнды при необходимости приводятся к численному типу.

mod

`mod` — деление по модулю, но в отличие от `%` приведение типов не выполняется. Сравните четыре примера.

Деление по модулю двух целых чисел выполняется одинаково:

```
say 10 % 3;   # 1
say 10 mod 3; # 1
```

Если один из операндов сделать строкой, то оператор `%` приведет ее к числу:

```
say 10 % "3"; # 1
```

А при использовании оператора `mod` возникнет ошибка:

```
say 10 mod "3";

Calling 'infix:<mod>' will never work with argument types (Int,
    Str)
    Expected any of:
    :(Real $a, Real $b)
```

Поэтому преобразование необходимо выполнить явно:

```
say 10 mod +"3"; # 1
```

Или:

```
say 10 mod "3".Int; # 1
```

%%

%% — оператор, сообщающий о возможности целочисленного деления (divisibility, «делибельность») без остатка. Возвращает булевое значение.

```
say 10 %% 3; # False
say 12 %% 3; # True
```

+&, +|, +^

+&, +|, +^ — побитовое умножение, сложение и XOR. Плюс в операторе намекает на то, что операнды перед выполнением действия приводятся к типу `Int`.

?|, ?&, ?^

?|, ?&, ?^ приводят операнды к логическому типу и выполняют логические операции OR, AND и XOR.

+<, +>

+<, +> — операторы побитового сдвига влево или вправо.

```
say 8 +< 2; # 32
say 1024 +> 8; # 4
```

gcd

gcd (greatest common denominator) вычисляет наибольший общий делитель двух чисел.

```
say 50 gcd 15; # 5
```

lcm

`lcm` (least common multiple) находит наименьшее общее кратное.

```
say 1043 lcm 14; # 2086
```

==, !=

==, != выполняют сравнение двух численных операндов. Данные приводятся к типу `Numeric` при необходимости.

<, >, <=, >=

<, >, <=, >= — операторы для численного сравнения.

<=>

<=> — оператор для сравнения чисел, возвращающий значение типа `Order`: `Order::Less`, `Order::More` или `Order::Same`.

Операторы для работы со строками

~

~ — конкатенация строк. Точка в Perl 6 теперь используется для вызова методов, а все действия, связанные со строками, используют тильду.

```
say "a" ~ "b"; # ab
```

При необходимости происходит преобразование типов:

```
say "N" ~ 1; # N1
say 4 ~ 2;   # 42
```

x

x повторяет строку указанное число раз.

```
say "A" x 5; # AAAAA
```

Нестроковые значения перед повторением будут преобразованы в строку:

```
say 0 x 5; # 0000
```

Если запрошено отрицательное число повторов, возвращается пустая строка.

eq, ne

eq, ne сравнивают строки (или приведенные к строке значения) на равенство и неравенство, соответственно.

lt, gt, le, ge

`lt`, `gt`, `le`, `ge` — операторы для сравнения строк: меньше, больше, меньше или равно, больше или равно. Операнды приводятся к строкам.

leg

`leg` — оператор, сообщающий, равны ли строки, либо одна из них «больше» или «меньше» (в алфавитном порядке). Аналогична оператору `cmp` из Perl 5 (но не из Perl 6), но возвращает значение `Order::Less`, `Order::More` или `Order::Same`.

```
say "a" leg "b";        # Less
say "abc" leg "b";      # Less
say "bc" leg "b";       # More
say "abc" leg "ABC".lc; # Same
```

Перед сравнением все операнды приводятся к строкам.

```
say 42 leg "+42"; # More
say 42 leg "42";  # Same
```

Универсальные операторы сравнения

В Perl 6 определены несколько операторов, которые одинаково подходят для сравнения как строк, так и для чисел и даже для составных объектов (например, пар).

cmp

`cmp` сравнивает два объекта и возвращает значение типа `Order` — одно из `Less`, `Same` и `More`.

```
say 2 cmp 2;   # Same
say 2 cmp 2.0; # Same
say 1 cmp 2;   # Less
say 2 cmp 1;   # More

say "a" cmp "b";         # Less
say "abc" cmp "b";       # Less
say "bc" cmp "b";        # More
say "abc" cmp "ABC".lc; # Same

my %a = (a => 1);
my %b = (a => 1);
say %a cmp %b; # Same
```

При сравнении величин разных типов (строки с числом, например) следует быть осторожным и помнить, что в языке определены несколько мультивариантов оператора:

```
proto sub infix:<cmp>(Any, Any) returns Order:D is assoc<none>
multi sub infix:<cmp>(Any,       Any)
multi sub infix:<cmp>(Real:D,    Real:D)
multi sub infix:<cmp>(Str:D,     Str:D)
multi sub infix:<cmp>(Enum:D,    Enum:D)
multi sub infix:<cmp>(Version:D, Version:D)
```

(`:D` в определении — это не смайл, а указание на то, что аргумент должен быть определен, то есть в переменной этого типа должно содержаться значение.)

Поэтому при сравнении строки с числом компилятор скорее всего выберет вариант функции с сигнатурой `(Str:D, Str:D)`, и оба операнда будут рассматриваться как строки:

```
say "+42" cmp +42; # Less
say ~42 cmp +42;   # Same
```

Обратите внимание на отличие этого оператора от одноименного, доступного в Perl 5. Ср. с оператором `leg`.

before, after

`before`, `after` — универсальные операторы сравнения, работающие с числами, строками и объектами других типов. Возвращает логическое значение `True` или `False` в зависимости от того, какой из операндов стоит раньше или позже.

Работа с преобразованиями типов операндов аналогична оператору `cmp`. Стоит иметь в виду, что в зависимости от типа данных сравнение либо чисел, либо «таких же» строк в разных случаях может привести к противоположным результатам, поскольку числа сравниваются как числа, а строки — как строки в алфавитном порядке:

```
say 10 before 2;      # False
say '10' before '2';  # True

say 10 before 20;     # True
say '10' before '20'; # True
```

eqv

`eqv` — оператор, сравнивающий объекты на эквивалентность. Возвращает булевое значение, истинное в том случае, если оба операнда имеют одинаковый тип и содержат одинаковые данные.

```
my $x = 3;
my $y = 3;
say $x eqv $y; # True
```

Пример с более сложными данными:

```
my @a = (3, 4);
my @b = (3, 4, 5);
@b.pop;
```

```
say @a eqv @b; # True
```

Поскольку целые числа, числа с плавающей точкой и строки, содержащие только цифры, относятся к разным типам данным, следующие сравнения дадут `False`:

```
say 42 eqv 42.0; # False
say 42 eqv "42"; # False
```

Важно не попасть в ловушку, когда в дело вступает тип `Rat`:

```
say 42 eqv 84/2;       # False
say 42 eqv (84/2).Int; # True
```

===

`===` — оператор, возвращающий истинное значение в том случае, если операнды являются одним объектом, и ложное в остальных случаях.

```
class I {
}

# Три разных экземпляра.
my $i = I.new;
my $ii = I.new;
my $iii = I.new;

my @a = ($i, $ii, $iii);
for @a -> $a {
    for @a -> $b {
        say $a === $b; # Печатает True только там, где $a и $b
                       # указывают на один и тот же элемент
                       # массива @a.
    }
}
```

=:=

`=:=` — оператор для проверки того, что операнды ссылаются на один и тот же объект. Возвращается истина или ложь.

```
my $x = 42;
my $y := $x;

say $x =:= $y; # True
say $y =:= $x; # True
```

~~

~~ — оператор смартматчинга. Оператор выполняет сравнение объектов и на первый взгляд работает независимо от их типа, например:

```
say 42 ~~ 42.0; # True
say 42 ~~ "42"; # True
```

Оператор не является коммутативным, и может возвращать разные результат при перестановки операндов:

```
say "42.0" ~~ 42; # True
say 42 ~~ "42.0"; # False
```

Это поведение проясняется, если рассмотреть алгоритм работы оператора ~~. Оператор вычисляет значение правого операнда и вызывает на нем метод ACCEPTS, которому передает значение левого операнда (точнее, ссылку $_ на него). Для каждого типа данных существует свой вариант метода ACCEPTS. Для строк, например, он сравнивает строки, а для чисел — числа.

Предыдущие два примера эквивалентны следующим:

```
say 42.ACCEPTS("42.0"); # True
say "42.0".ACCEPTS(42); # False
```

Операторы для работы со списками

xx

xx повторяет список заданное число раз.

```
say((1, −1) xx 2); 1 −1 1 −1
```

Аналогично строковому x, оператор xx возвращает пустой список, если число повторов отрицательное или нулевое.

Z

Z — зип-оператор (zip). Смешивает два массива, возвращая по очереди элементы то одного, то другого из них до тех пор, пока в каждом из списков достаточно элементов.

Запись

```
1 @c = @a Z @b;
```

эквивалентна примерно следующей:

```
1 @c = (@a[0], @b[0], @a[1], @b[1], ...);
```

X

X — оператор, формирующий из двух списков третий, состоящий из всех возможных комбинаций элементов исходных массивов.

```
1 @c = @a X @b;
```

То же, что:

```
1 @c = (@a[0], @b[0], @a[0], @b[1], @a[0], @b[2], ..., @a[N], @b
    [0], @a[N], @b[1] ..., @a[N], @b[M]);
```

Операнды не обязательно должны быть одинаковой длины.

...

... — оператор, создающий ленивые списки.

```
1 my @list = 1 ... 10;
```

В отличие от .., три точки умеют считать и в обратную сторону:

```
my @back = 10 ... 1;
```

Звездочка в качестве конечного элемента создаст бесконечный ленивый список:

```
my @long = 1 ... *;
```

Операторы для объединений

|, &, ^

|, &, ^ — операторы, создающие junctions (объединения? они же ранее известны как «квантовые суперпозиции»). Эти объекты могут использоваться на месте скаляров, но ведут себя одновременно как несколько значений.

Операторы |, & и ^ создают, соответственно, junctions типа any, all и one. Junctions ведут себя как скаляры, содержащие *одновременно* несколько значений:

```
# Значение 4 – одно из перечисленных:
say "ok" if 4 == 1|2|3|4|5;

# 4 нет ни в каком из указанных:
say "ok" if 4 != 1 & 2 & 3 & 5;

# 4 повторяется дважды, поэтому оно не единственное:
say "ok" unless 4 == 1 ^ 2 ^ 2 ^ 4 ^ 4 ^ 5;
```

Shortcut-операторы

&&

&& возвращает первый из операндов, который будучи преобразованным в логический тип становится False. Обратите внимание, что возвращаемое значение — не True или False, а оригинальное значение одного из операндов (если конечно он уже не был False).

```
say 10 && 0; # 0
say 0 && 10; # 0;

say 12 && 3.14 && "" && "abc"; # пустая строка
```

Как только найдено подходящее значение, работа оператора закончится, и все оставшиеся справа значения вычисляться не будут.

||

`||` возвращает первый операнд, который в логическом контексте является `True`. Последующие значения не вычисляются.

```
say 10 || 0; # 10
say 0 || 10; # 10
```

^^

`^^` — возвращает операнд, который является в булевом контексте истинным, и при этом он такой единственный. Если ничего не найдено, возвращается `Nil`. Как только найден второе истинное значение, вычисления оставшихся прекращаются.

```
say 0 ^^ '' ^^ "abc" ^^ -0.0;  # abc
say 0 ^^ '' ^^ "abc" ^^ -10.0; # Nil
```

//

`//` возвращает первый определенный (defined) операнд. Остальные при этом не вычисляются.

```
my $x;
my $y = 42;
my $z;
say $x // $y // $z; # 42
```

Другие инфиксные операторы

min, max

`min`, `max` возвращают, соответственно, минимальный и максимальный из своих операндов.

?? !!

`?? !!` — тернарный условный оператор. Работает как `? :` в Perl 5.

```
1 say rand < 0.5 ?? 'Yes' !! 'No';
```

=

`=` — оператор присваивания.

=>

`=>` — конструктор пар. Создает пары ключ — значение, причем ключ не обязательно заключать в кавычки.

```
1 my $pair = alpha => "one";
2
3 my %data = jan => 31, feb => 28, mar => 31;
```

,

`,` — оператор конструирования парселов (parcel).

```
1 my $what = (1, 2, 3);
2 say $what.WHAT; # (Parcel)
```

При вызове функций запятая используется как разделитель передаваемых параметров.

:

: используется при вызове методов как разделитель аргументов. Понять работу двоеточия можно на следующем примере:

```
class C {
    method meth($x) {
        say "meth($x)";
    }
}
my $o = C.new;
meth($o: 42); # Вызывается метод meth объекта $o, печатается meth
    (42)
```

После инфиксного двоеточия обязательно должен быть пробел (если его нет, двоеточие может означать именованный параметр), и оно должно стоять только после первого аргумента.

■ *Андрей Шитов*

4. Метаоператоры в Perl 6

Рассмотрены доступные в языке метаоператоры — операторы, расширяющие синтаксические возможности языка, используя другие операторы

Дизайн операторов в Perl 6 сделан максимально регулярным, поэтому, например, при добавлении нового оператора Perl 6 сам создаст еще несколько, чтобы привести в порядок гармонию. Для этого созданы метаоператоры — операторы над операторами. В этой статье кратко описаны все доступные метаоператоры, их восемь.

Присваивание

Метаоператор присваивания (=) создает из других операторов операторы вида +=, ~= и т. д. Действие, выполняемое таким оператором, всегда эквивалентно более многословной записи.

Конструкция `$a op= $b` с оператором op выполняет то же самое, что `$a = $a op $b`.

То есть `$x += 2` эквивалентно `$x = $x + 2`, а `$str ~= '.'` — `$str = $str ~ '.'`.

Другие варианты менее очевидны, однако работают строго в соответствии с определением. Пример такого оператора: `,=`, он добавит новые элементы к существующему списку:

```
1 my @a = 1..5;
2 @a ,= 6;      # то же, что @a = @a, 6;
3 say @a;       # 1 2 3 4 5 6
```

Теперь посмотрим, как проявляется регулярность при добавлении нового оператора, сколько угодно нестандартного, например, такого:

```
1 sub infix:<^_^>($a, $b) {
2     $a ~ '_' ~ $b
3 }
```

При обычном использовании оператор объединяет свои два операнда:

```
say 4 ^_^ 5; # 4_5
```

Но при этом автоматически заработала и форма ^_^=:

```
my $x = 'file';
$x ^_^= 101;
say $x; # file_101
```

Отрицание

Метаоператор, выполняющий булево отрицание, — восклицательный знак !. Соответственно, если оператор op возвращает булевое значение, то он может быть расширен до формы !op.

```
say "no" if "abcd" !~~ "e";
```

Обратный (reverse) оператор

Оператор, принимающий два операнда (то есть любой инфиксный оператор, начиная со / и заканчивая cmp) получает форму с метапрефиксом R, который обменивает операнды местами, одновременно при необходимости изменяя ассоциативность на противоположную (то есть если в форме $a op $b op $c сначала выполняется действие над $a и $b, то в форме $a Rop $b Rop $c первым будет вычислено выражение $c op $b).

```
say 2 R/ 10; # 5. То же, что say 10 / 2
```

Более ощутимую пользу метаоператор R приносит, по-видимому, в операторах редукции, например:

```
say [R~] 'a'..'z'; # zyxwvutsrqponmlkjihgfedcba
```

Редукция (reduction)

Для любого инфиксного оператора op будет работать его вариант [op]. Такой оператор может принимать список, а выполнение эквивалентно записи, в которой между элементами списка стоял бы оператор op. Например:

```
[*] 1..5
```

является сокращенной записью для

```
1 * 2 * 3 * 4 * 5
```

Все это действует и для операторов, которые определил сам программист, например, заработает редуцированная форма с созданным ранее оператором `^_^`:

```
say [^_^] 1..10; # 1_2_3_4_5_6_7_8_9_10
```

Кросс-операторы

Кросс-метаоператор `X` применяет операцию ко всем возможным парам своих аргументов-списков. В результате применения кросс-оператора получается список.

```
say 'a'..'h' X~ 1..8;
```

Этот пример напечатает все номера полей шахматной доски.

Зип-оператор (zip)

Зип-метаоператор `Z`, аналогично кросс-оператору, совмещает аргументы из двух списков, но при этом выполняет операцию только между соответствующими друг другу элементами, игнорируя лишние. Иными словами, запись

```
@a Z+ @b
```

равносильна такой:

```
(@a[0] + @b[0], @a[1] + @b[1], . . ., @a[*−1] + @b[*−1])
```

Примечание. При попытке обратиться к последнему элементу массива через индекс −1, как это возможно в Perl 5, компилятор предложит вариант, доступный в Perl 6:

```
Unsupported use of a negative −1 subscript to index from the end;
in Perl 6 please use a function such as *−1
```

Гипероператоры

Гипероператоры полезны для работы со списками и модифицируют обычные операторы таким образом, что он применяется ко всем элементам массива или списка. Модифицироваться могут и унарные, и бинарные операторы. Чтобы получить гипероператор, достаточно добавить к оператору последовательность >> и/или <<.

Например, для унарного оператора:

```
my @a = (True, False, True);
my @b = !<< @a;
say @b; # False True False
```

Вариант с постфиксным оператором:

```
my @a = (1, 2, 3);
@a>>++;
say @a;
```

Здесь важно следить за пробелами. Если оригинальный оператор разрешает поставить пробел, то он допустим и в гиперформе. А если нет, то наличие пробела будет синтаксической ошибкой.

```
my @a = ('a', 'b')>>.uc; # Здесь нельзя написать ('a', 'b') >>.
    uc
say @a; # A B
```

Для бинарных операторов требуется две пары угловых скобок. При этом возможны четыре варианта со скобками, направленными в разные стороны:

```
>>+>>
<<+<<
<<+>>
>>+<<
```

Выбор формы зависит от желаемого результата. Если с обеих сторон стоят массивы одинаковой длины, подойдут симметричные формы:

```
my @a = (1, 2, 3) >>+<< (4, 5, 6);
say @a; # 5 7 9
```

Или так:

```
my @a = (1, 2, 3) <<+>> (4, 5, 6);
say @a; # 5 7 9
```

Если же один из операндов содержит меньше элементов, чем другой, то форма гипероператора влияет на то, нужно ли сообщить об ошибке, либо размножить более короткий операнд так, чтобы на каждый элемент более длинного пришлось по элементу из повторяющейся последовательности элементов короткого.

Например, в простом случае со скаляром:

```
say((1, 2, 3) >>+>> 1); # 2 3 4
```

Здесь единица справа повторится три раза, поэтому каждый элемент списка `(1, 2, 3)` будет увеличен на единицу.

Аналогичный пример для двух массивов:

```
my @a = (1, 2, 3, 4) >>+>> (1, −1);
say @a; # 2 1 4 3
```

Фактически выполняется поэлементное сложение `(1, 2, 3, 4)` и `(1, −1, 1, −1)`. Острой стороной стрелки направлены в сторону операнда, который будет размножен.

Аналогично в обратном порядке:

```
my @b = (1, −1) <<+<< (1, 2, 3, 4);
say @b; # 2 1 4 3
```

Если заранее неизвестно, с какой стороны больше данных, но при этом в любом случае требуется выполнить операцию над всеми элементами более длинного списка, можно воспользоваться формой `<<+>>`:

```
my @a = (1, −1) <<+>> (1, 2, 3, 4);
say @a; # 2 1 4 3

my @b = (1, 2, 3, 4) <<+>> (1, −1);
say @b;  # 2 1 4 3
```

Оператор же `>>+<<` ожидает, что размер массивов по обе стороны одинаковый. Если использовать его в предыдущих двух примерах, компилятор сообщит об ошибке:

```
Lists on either side of non-dwimmy hyperop of infix:<+> are not
    of the same length
left: 2 elements, right: 4 elements
```

Наконец, для гипероператоров предусмотрена альтернативная запись с использованием юникодных кавычек, например:

```
say((1,2) »+« (3,4));
```

(Если не ошибаюсь, исторически сначала появились именно юникодные варианты, а затем были добавлены ASCII-эквиваленты.)

Домашнее задание читателю: в чем отличие `@a >>+<< @b` от `@a Z+ @b`?

Последовательности

Метаоператор `S` — инструкция компилятору о том, что все действия нужно выполнять последовательно, без попыток срезать углы или распараллелить вычисления. Вычисления должны производиться именно в том порядке, который явно указан в программе.

Например, оператор `S&` обязан вычислить все части выражения, даже если результат будет известен раньше.

Наглядный пример, показывающий влияние метаоператора `S`:

```
sub f() {
    say "f()";
    return False;
}

sub t() {
    say "t()";
    return True;
}

my $y = f() & t(); # f()
say ?$y;

my $z = f() S& t(); # f(), t()
say ?$z;
```

При вычислении `f()& t()` будет вызвана только первая функция, которая сразу же сделает бессмысленным вычисление второй, однако во втором случае `f()S& t()` вычисляется и правый операнд. (Rakudo 2015.03 печатает во втором случае True, что неверно.)

Примечание: одиночный амперсанд создает `Junction`, поэтому требуется преобразование в логический тип.

Аналогично, метаоператор `S` должен запрещать распараллеливать вычисления в гипер-, кросс- и редуцированных операторых (но пока и само распараллеливание не реализовано).

Заключение

Описанные в статье возможности Perl 6 выходят за рамки того, к чему программист привык по опыту работы с Perl 5. Какие-то вещи выглядят на первый взгляд странными и нелепыми. Предлагаю читателям подумать над полезными применения метаоператоров — поделитесь своими мыслями в комментариях.

■ *Андрей Шитов*

5. Обзор CPAN за апрель 2015 г.

Рубрика с обзором интересных новинок CPAN за прошедший месяц.

Статистика

- Новых дистрибутивов — 226
- Новых выпусков — 822

Новые модули

JSON::XS::Sugar

`JSON::XS::Sugar` — это небольшой синтаксический сахар при работе с JSON::XS. Модуль экспортирует константы `JSON_TRUE` и `JSON_FALSE`, а также функции `json_truth`, `json_number`, `json_string`, которые приводят аргумент к нужному типу.

Photography::Website

Утилита `photog` позволяет сгенерировать из каталога с фотографиями статический сайт-фотогалерею с фотографиями отсортированными в хронологическом порядке с поддержкой вложенных альбомов. Возможно задание своего шаблона и добавление водяного знака в изображения.

App::PAUSE::TimeMachine

Веб-сервис и утилита командной строки для извлечения файла со списком пакетов PAUSE на заданный момент времени. Модуль был создан во время Берлинского QA-хакатона. Для доступа к истории файла 02packages.details.txt используется git-репозиторий https://github.com/batchpause/PAUSE-git

Carmel

`Carmel` — это новый менеджер пакетов CPAN, основанный на «артефактах» cpanm. В отличии от всех других инсталляторов, carmel сохраняет каталоги-артефакты, в которых происходила сборка пакетов в едином репозитории (по умолчанию это каталог `$HOME/.carmel/{version}-{archname}/build`). Carmel сканирует этот каталог в поисках сборочных каталогов пакетов нужных версий и формирует соответствующий `@INC`.

Обновлённые модули

Devel::NYTProf 6.01

Вышел новый мажорный релиз модуля для профилирования `Devel::NYTProf`. В новом релизе множество улучшений работы модуля на платформе Windows, в частности, использование часов высокого разрешения `QueryPerformanceCounter`. Улучшена документация модуля.

Module::Signature 0.78

Важное обновление `Module::Signature`, который используется многими CPAN-клиентами для проверки GPG-подписи модулей, полученных со CPAN или любого из его зеркал. В версии 0.75 было исправлено четыре уязвимости в коде модуля:

- CVE-2015-3406. `Module::Signature`, из-за некорректного парсинга границ подписанных данных в файле подписи, мог извлечь информацию о файлах из неподписанной части.
- CVE-2015-3407. При проверке содержимого CPAN-модуля `Module::Signature` игнорировал некоторые файлы из архива, которые не были указаны в файле подписи, что затрагивает и файлы в каталоге t, которые автоматически выполняются при запуске тестов.
- CVE-2015-3408. При генерации контрольных сумм из подписанного манифеста `Module::Signature` использовал `open()` с двумя параметрами

при чтении файлов. Это позволяло встраивать произвольные shell-команды в файле SIGNATURE, которые бы запускались при проверке подписи.
- CVE-2015-3409. `Module::Signature` подгружает некоторые модули в процессе работы, модуль `Text::Diff` может отсутствовать в системе и, соответственно, может быть загружен из текущего каталога распакованного архива модуля, что может привести к выполнению произвольного кода.

BSON 0.12

После долгого перерыва обновился модуль (де)сериализации формата BSON. Добавлена поддержка типа 0x06 (Javascript “undefined”), который как и null значение декодируется в `undef`. Ускорена работа модуля в среднем на 10-20% за счёт встраивания функций.

libintl-perl 1.24

Вышло обновление библиотеки интернационализации libintl-perl. Изменилась лицензия, под которой выпускается исходный код: вместо LGPL 2.1 стала GPL 3. Также исправлено несколько ошибок, включая исправление для работы на системах без поддержки локалей, например, Android.

Mail::SpamAssassin 3.4.1

Обновлён детектор спама `Mail::SpamAssassin`. В новой версии улучшена автоматизация в борьбе со спамерами, которые используют новые домены верхнего уровня. Используется нормализация текста, чтобы упростить создание правил против спамеров, которые используют альтернативные кодовые наборы, чтобы обойти тесты. Продолжается работа над совершенствованием нативной поддержки IPv6. Улучшена Байесова классификация с улучшенной отладкой и хешированием вложений.

Validate::Tiny 1.501

Минималистичный модуль для проверки параметров `Validate::Tiny` имеет несколько несколько несовместимых изменений: метод `new` теперь просто создаёт объект, для выполнения проверки необходимо вызвать метод `check`, удалена функция `error_string`.

XML::LibXML 2.0119

Вышло обновление безопасности для `XML::LibXML` с исправлением CVE-2015-3451. `XML::LibXML` в некоторых случаях игнорировал опцию `expand_entities` , что могло приводить к утечке данных.

■ *Владимир Леттиев*

6. Интервью с Сюзанной Шмидт

Сюзанна Шмидт (sushee) — политолог, которая стала Perl-программистом

Когда и как научились программировать?

Хм… Я начала изучать программирование, когда мне было уже 26 лет и я уже почти закончила университет по специальности политолог.

В 1994 меня зацепили компьютеры и Linux, когда мой друг рассказал мне про свободный софт и интернет, поэтому я забросила политологию и переключилась на компьютеры.

В Берлине было по-особенному, потому что интернет, веб и Linux появились примерно в одно и тоже время, и я пыталась за всем уследить.

Я почти ничего не знала о компьютерах, кроме как писать статьи в Ворде DOS. Но у меня появился компьютер, и я установила — совершенно не понимая, что происходит — Linux с помощью немецкого IRC-канала. Но мне хотелось научиться программировать.

Поэтому я пыталась найти понятные для себя книги по программированию и, конечно, все говорили совершенно разное, с чего же мне стоит начать. «Pascal! Он для обучения!», «C! Это Unix!», «Lisp! Потому что это единственный настоящий язык» и так далее, и тому подобное.

В то время книги были написаны в основном для ученых, поэтому у меня не получалось чему-то из них научиться, программирование было для меня чем-то чужим, мне не далась даже K&R — я просто ее не поняла.

Учитывая то, что веб только начал развиваться, я помню интернет по протоколу gopher и тому подобное :) Я сделала домашнюю страничку. На HTML. Потом я нашла книгу Perl 4 издательства O'Reilly и вскоре после этого «Programming Perl», и тогда это была ТА книга, которая рассказывала о «есть такой список вещей» и «есть еще контекст» (я политолог и, конечно, У ВСЕГО ЕСТЬ СВОЙ КОНТЕКСТ, А КАК ЖЕ ИНАЧЕ? ;), и неожиданно все стало понятным.

Когда и как познакомились с Perl?

Perl был поим первым языком, и на самом деле благодаря стилю, в котором была написала книга, мне удалось понять, как использовать циклы и регулярные выражения для переформатирования большого документа… поэтому я познакомилась с Perl и регулярными выражениями до всего остального.

Веб только начинался, и первой моей работой было администрирование (связанное с Linux) и веб-разработка (HTML и Perl), и постепенно я продвигалась вверх, как и многие из моего поколения.

Какой редактор используете?

С самого начала Vi и немножко pico, благодаря elm ;) Также некоторое время я пользовалась emacs, но вернулась на vi/vim и никогда об этом не пожалела. (И да, у меня тоже не получалось из него выйти, я просто выдергивала модемный кабель и переподключалась… ;)

С какими другими языками интересно работать?

В какой-то момент я заметила, что мне хорошо дается SQL, и что мне очень нравятся регулярки, поэтому я обратила внимание на Prolog (почитайте Wikipedia-статью про декларативные языки). Мне нравится шелл, нравится минимализм C и ассемблера, мне нравится продуктивность R и также нравится привлекательность Ruby. В конце концов мне нравятся небольшие языки, в особенности минималистичные, такие как Scheme, например.

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

Он не стоит на пути как другие языки. Я часто замечаю, что у Perl есть своеобразная мягкость, с которой он соединяет вещи между собой. Мне нравится, что на Perl можно писать в разном стиле — ООП (здесь я имею в виду определение ООП, данное Аланом Кейем — отправка сообщений…) или функциональном, как угодно. До недавнего времени у CPAN было сильное преимущество, но сейчас у каждого языка есть нечто подобное.

Что, по-вашему, является самым важным свойством языков будущего?

Простота расширения языков, как хорошо взаимодействуют между собой компоненты и сторонние модули. Насколько они поддаются модификации. Возможно, как хорошо языки поддерживают функциональную парадигму. Мне кажется, это и есть будущие тенденции.

Что думаете о Perl 6?

Мне очень грустно. Я была так воодушевлена в… 200* и писала оды замечательным возможностям Perl 6… но он сильно перегружен, очень запутанный и сложный язык. Мне хотелось лучшего Perl. И конечно, его выход сильно затянулся, чтобы стать чем-то важным — на сегодняшний день есть столько изящных и замечательных языков, зачем вообще смотреть на Perl 6? Несколько раз я пыталась, но это просто не для меня. Я придерживаюсь современного минималистичного стиля Perl 5, и мне это очень нравится (например, Mojolicious или модули LeoNerd) или смотрю на другие языки.

Что для вас сообщество программистов? Считаете ли вы себя частью сообщества Perl?

Обычно это группа единомышленников и сочувствующих — те кто пишет код или что-то с большой пользовательской базой (как, например, у R). Эти люди живут в особой культуре, и обычно это и есть «сообщество».

Я считаю себя лишь наполовину в Perl сообществе, мне стоит усилий не зацикливаться только на Perl и заниматься другими вещами. Я не хочу по прошествии времени обнаружить только Perl у себя в резюме.

Где сейчас работаете, сколько времени проводите за написанием Perl-кода?

Половина времени это Postgres, половина время это Perl, потому что я занимаюсь бекендом :) Еще немного шелла, make, JavaScript там и сям.

Какие плюсы и минусы работы из дома?

Пожалуй, я отвечу самым лучшим комиксом на эту тему:

http://theoatmeal.com/comics/working_home ;)

- Плюсы: концентрация и тишина, гибкость
- Минусы: концентрация и тишина, гибкость (раздражители :)

Стоит ли сейчас молодым программистам советовать учить Perl?

Да! Я считаю Perl «классикой», и мне кажется, что каждый должен знать классику: Unix, C, Smalltalk, Lisp, Perl и так далее.

Вопросы от читателей

Уже прошло три месяца с вашей регистрации на PAUSE, где загрузки? :)

Очень сильно занята, честно :) И на CPAN уже почти все есть, не нужно спешить, чтобы что-нибудь добавить.

Считаете ли вы, что постинг картинок котов важен для современной эры интернета?

ДА. Как иначе людям правильно расслабляться? И каждому нужна ежедневная доза няшек. ♥ ;)

■ *Вячеслав Тихановский*