0
Рейтинг

Evrone

Подписаться

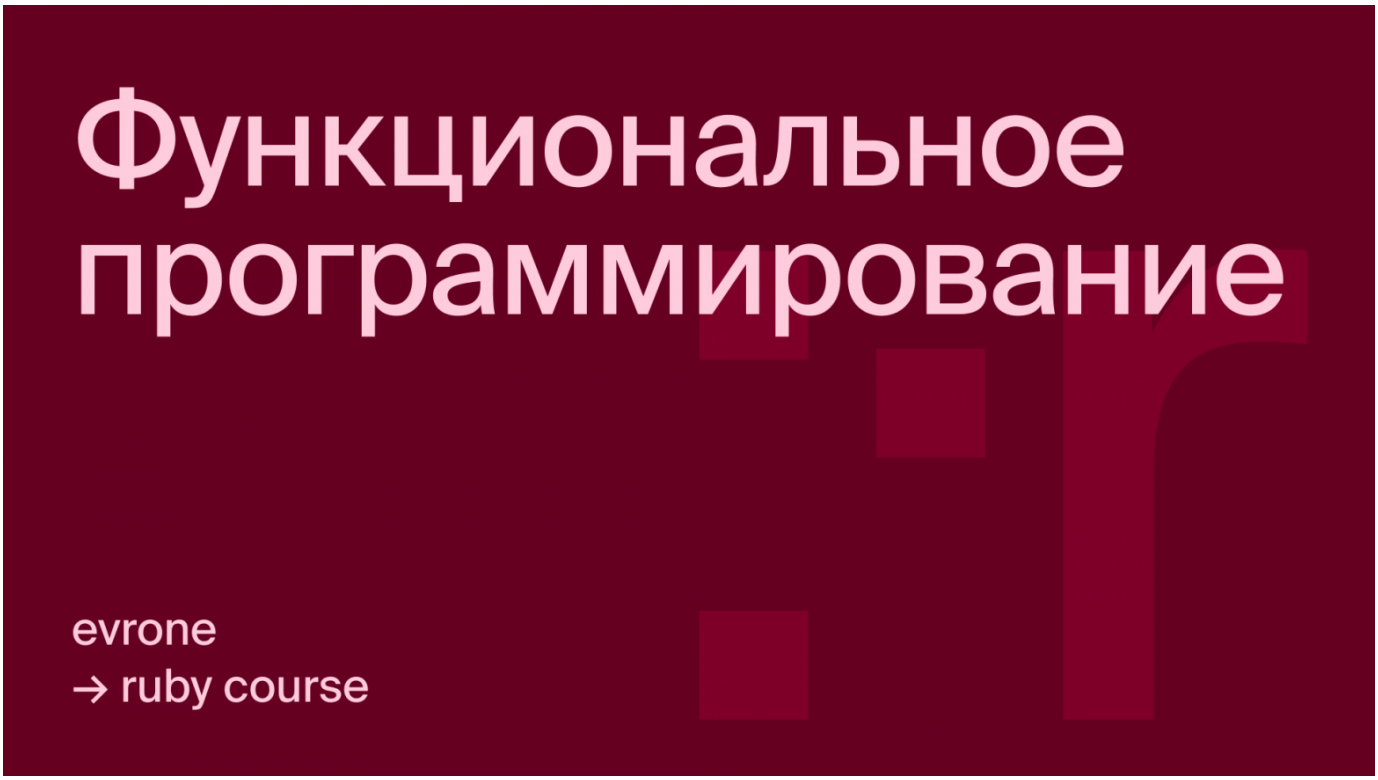
Evrone
13 окт 2022 в 17:23

Курс по Ruby+Rails. Часть 3. Функциональное программирование

11 мин 3.6K

Блог компании Evrone, Ruby*, Ruby on Rails*

Туториал



Привет! Сегодня мы поговорим про функциональное программирование. В Ruby реализован исключительно гибкий объектно-ориентированный стиль. И как бы он ни был хорош, он имеет свою цену: иногда для реализации сложных алгоритмов и систем он слишком гибкий.

В Ruby нет строгих соглашений о типах, он реализовывает и поощряет «утиную типизацию». Это удобно, но может обернуться некорректностью программы, если программист не будет самостоятельно тщательно следить за качеством кода.

Особенно это болезненно для алгоритмов, в которых надёжность важнее скорости прототипирования. Иногда (а на самом деле довольно часто) нам нужен код, который не будет подобен молотку, видоизменяющемуся и улучшающемуся прямо в процессе работы. В идеальной ситуации веб-программист должен создавать приложения, которые обрабатывают данные надёжно, понятно и логично.

В объектно-ориентированном стиле программирования привнесённую сложность принято снижать паттернами, практиками, принципами. Многие из них мы разберём в следующих лекциях. А сейчас посмотрим на стиль программирования, который был изначально придуман для создания логичных и изящных алгоритмов — функциональное программирование. Ruby поддерживает его «из коробки».

Функциональное программирование существует в мире программирования давно, правда, долгое время все его изучение было скорее заботой теоретиков. Со временем, однако, полезных знаний накопилось достаточно много, чтобы начать их использовать в программировании широкого профиля. Сегодня практически невозможно найти язык, в котором нет всех основных элементов функционального программирования.

Основные принципы функционального программирования

Давайте посмотрим на основные принципы и разберём примеры кода, которые реализуют довольно сложные операции. Мы изучим, какие в Ruby есть инструменты функционального стиля.

Итак, принципы:

- Во-первых, программа в функциональном стиле — это сложное выражение, в котором функции применяют функции. «Применение функций» — аналог термина «вызов процедур»

процедурного стиля. Функции применяют функции — запомним это как аксиому, чтобы понимать, как работает всё остальное.

- Во-вторых, функции должны быть чистыми. То есть код должен быть построен так, чтобы функция, применённая к одним и тем же аргументам, возвращала одно и то же значение, не важно где и как мы используем её. Чистая функция не изменяет своё окружение. Функции, которые изменяют окружение (например, выполняют ввод-вывод или мутируют, как методы, состояние своего объекта), выделяются в специальные участки кода, которые мы строго контролируем при помощи тестов.
- В-третьих, переменные, объекты и состояния не изменяются. Функция может сконструировать и вернуть новый объект, но не станет изменять переданные ей аргументы. Этот принцип перекликается со вторым, оговорки и ограничения у него такие же (если нужно, умный компилятор сам сделает внутреннее представление программы наиболее эффективной, а программист будет работать с логически чистым кодом).

Эти три принципа составляют костяк ФП. Этот стиль ограничивает программиста в правах на произвольное изменение состояния системы. Взамен программист получает логически строгий и непротиворечивый код без «подводных камней», а машина, то есть рантайм языка программирования — возможность генерировать максимально эффективный, корректный и безопасный код с наименьшими затратами.

Давайте сделаем небольшое отступление. На самом деле каждый стиль программирования накладывает на программиста какие-то ограничения в обмен на уменьшение сложности и увеличение читаемости кода.

Процедурный стиль не позволяет собирать все инструкции в одну большую кучу, структурный — использовать `GOTO` и скакать по алгоритму, как захочется, объектно-ориентированный — выполнять над объектами действия, не предусмотренные чётко определёнными методами, функциональный — запрещает случайные мутации состояния объектов в отдельности и состояния системы в целом.

Посмотрите на простой пример, чтобы понять, как работают принципы функционального программирования на практике.

Функция `get_first` делает своё дело, однако её автор ошибся и забыл, что в Ruby в данном контексте `Array#shift` мутирует аргумент, и после каждого применения `get_first` первоначальный массив будет меняться:

```
def get_first(array)
  array.shift
end
```

Поддерживать такой код будет тяжело. Изменение массива придётся учитывать при дальнейшей разработке с этой функцией, документировать и тщательно тестировать состояние системы во избежание неожиданностей. Такое поведение функции называется «побочным эффектом» или просто «эффектом».

Реализуем её иначе на следующем примере:

```
arr = [1,2,3,4,5]

get_first(arr)
#> 1
arr
#> [2, 3, 4, 5]
```

Здесь `get_first` является чистой функцией: результат зависит только от аргумента, побочного эффекта нет. Мы учли пару несложных принципов, применили ФП в чистом виде, а взамен получили код без сюрпризов.

Инструменты ФП

Сами принципы, которые мы разобрали до этого, Ruby оставляет на совести программиста. Вместе с тем, для создания эффективных функциональных алгоритмов есть целый набор готовых инструментов. Мы пройдёмся по списку свойств, важных для программирования в функциональном стиле, и рассмотрим, как они реализованы в Ruby.

ФП предполагает, что функция — это самостоятельный полноценный объект, семантически равноправный со всеми остальными объектами языка (это называется first-class object). Она должна иметь возможность существовать как анонимный объект и присваиваться переменным. Ещё функции должны иметь возможность принимать другие функции в аргументах и возвращать тоже функции — в таком случае они называются функциями высшего порядка.

В Ruby описанные свойства предусмотрены конструкцией языка: выражения, блоки и возможность строить объекты с любым поведением, включая применение (в терминах функционального программирования функции не вызывают, а применяют) к предъявленным аргументам, возможность свободно присваивать эти объекты переменным/константам и передавать в качестве аргументов.

«Пощупать» функции высшего порядка мы можем с помощью блоков, которые рассматривали в лекции об императивном стиле как управляющие конструкции.

Посмотрите на пример. В Ruby блок может быть единственным или последним аргументом метода. В связке «метод-блок» он выступает в качестве анонимной функции, а метод, принимающий его как параметр — как функция высшего порядка:

```
# возвести все элементы массива в квадрат
[1, 2, 3, 4, 5].map { |x| x ** 2 }
#> [1, 4, 9, 16, 25]

# суммировать квадраты всех элементов массива
[1, 2, 3, 4, 5].map { |x| x ** 2 }.reduce(0) { |acc, x| x + acc }
#> 55
```

Кстати, реализация функций `map/reduce` для базовых коллекций — must have любого уважающего себя функционального языка, а в Ruby они определены в стандартном модуле `Enumerable` и могут использоваться для любых стандартных и пользовательских классов, использующих этот модуль — `Array`, `Hash` и так далее.

Давайте сделаем аналогичные методы вручную — мы тут же заметим несколько любопытных деталей:

```
class MyCollection
  def initialize(*args)   # превращение аргументов в массив объектов
    @collection = args
  end

  attr_reader :collection
  alias to_a collection   # теперь у нас есть конвертор в массив — `to_a`

  def <<(element)         # а теперь — оператор `<<` — добавление в конец
    collection << element
  end

  # два след. метода реализуем максимально наивно — императивно

  def map
    new_collection = MyCollection.new
    for x in collection
      new_collection << yield(x)
    end
    new_collection
  end
```

```
    def reduce(acc_init)
      acc = acc_init
      for x in collection
        acc = yield(acc, x)
      end
      acc
    end
  end

  coll = MyCollection.new(1, 2, 3, 4, 5)
  #> #<MyCollection:0x00007f876b8db6f8 @collection=[1, 2, 3, 4, 5]>

  coll.map { |x| x ** 2 }
  #> #<MyCollection:0x00007f876b4c7120 @collection=[1, 4, 9, 16, 25]>

  coll.reduce(0) { |acc, x| acc + x }
  #> 15

  coll.map       { |x| x ** 2 }
      .reduce(0) { |acc, x| acc + x }
  #> 55
```

На примере – минимальный, «наивный» `map/reduce`. Наша версия отличается от классической использованием итерации вместо рекурсии в методах `map` и `reduce` и использованием императивного типа `Array` вместо функционального списка. К тому же, мы «заперли» императивную семантику внутри чистых функций, поэтому наш подход можно в целом считать функциональным.

Блок играет роль анонимной функции, передаваемой в функцию высшего порядка в качестве аргумента. Внутри функции высшего порядка вызов `yield` применяет анонимную функцию-блок к аргументам `yield`, которые становятся значениями параметров в блоке.

У `yield` есть одна интересная особенность. Обратите внимание: блок не вычисляется немедленно при вызове применяемого к нему метода. Он «ждёт», пока его выполнение понадобится вызывающему методу, и выполняется только в момент передачи ему управления в вызове `yield`. В таком случае говорят, что блок имеет ленивую семантику – значение вычисляется только в момент востребования. Ленивая семантика – одна из отличительных черт функционального стиля. Аналогичным образом работают все блоки в Ruby.

Итак, анонимные функции-блоки – первый функциональный примитив, который мы встречаем в Ruby.

А что, если нам нужны функции высшего порядка, которые принимают произвольное количество анонимных функций, а не только последней в списке, да и ещё и синтаксически обособленной? Могут ли блоки быть совершенно самостоятельными объектами, а не просто синтаксическим дополнением к умным методам?

И тут мы перейдём к классу `Proc`.

Повнимательнее рассмотрим метод, принимающий блок. В примере мы пользуемся синтаксисом, явно выделяющим блок в аргументах. Заодно вытаскиваем `map` из нашего специального класса и играем с ним как с отдельной функцией:

```
def map(array, &block)
  puts 'The block is:'
  p block

  new_array = []
  for x in array
    new_array << yield(x)
  end
  new_array
end

map([1, 2, 3, 4, 5]) { |x| x ** 2 }
# The block is:
# <Proc:0x00007f876b314df0 (pry):111>
#> [1, 4, 9, 16, 25]
```

Ruby верен своей семантике: блок — это самостоятельный объект, экземпляр класса `Proc`. Класс `Proc` определяет объекты с методом `call` — то есть, функциональные объекты. В ФП-стиле они, как можно понять, идентичны анонимным функциям. Посмотрите на пример, `proc`'и можно создавать из блоков специально:

```
proc { |x| x ** 2 }
#> #<Proc:0x00007f876b377bd0 (pry):112>
```

Образующиеся функции можно присваивать переменным и передавать аргументами в функции высшего порядка — то есть, `proc`'и — это объекты первого класса. Это мы видим на иллюстрации:

```
f = proc { |x| x ** 2 }
#> #<Proc:0x00007f876b42e3a8 (pry):113>

f.call(4)
#> 16

# альтернативный синтаксис вызова proc-а
f[4]
#> 16
```

Для ряда классов определён метод `to_proc`. В повседневной практике мы пользуемся его реализаций в классе `Symbol` для сокращения кода, как в примере:

```
(3 + 2).then { |x| x.to_s }
```

С только что изученным синтаксисом можно без изменений использовать написанный нами ранее пример `map/reduce`. Например, так:

```
coll
#> #<MyCollection:0x00007f876b8db6f8 @collection=[1, 2, 3, 4, 5]>

coll.reduce(0, &:+)
#> 15
```

Давайте объясню, как это работает. При наличии оператора `&` в аргументе, Ruby выполняет метод `to_proc` символа `:+`, который ищет подходящий по имени метод в текущем лексическом контексте и формирует `proc` с требуемыми аргументами и вызовом найденного метода. В контексте нашего `reduce` это создаёт блок вида `{ |x, y| x + y }`, который и выполняется в нужном месте. Проверим эту семантику отдельно от использующего блок метода:

```
plus = :+.to_proc
=> #<Proc:0x00007f876b40df68(&:+) (lambda)>

plus[1, 2]
=> 3
```

Лямбды

Вы заметили слово `lambda` в описании proc-объекта в предыдущем примере?

Слово «лямбда» пришло в программирование из лямбда-исчисления — изобретения математика и логика Алонзо Чёрча, который вместе с Аланом Тьюрингом считается одним из отцов информатики. Лямбды — это разновидность proc-объектов с некоторыми особенностями. Давайте их рассмотрим на примере:

```
f_proc = proc { |x| puts "=> #{x}" }
f_lambda = lambda { |x| puts "=> #{x}" }

f_proc.class
#> Proc
f_lambda.class
#> Proc

[54] pry(main)> f_proc.call
# =>
#> nil
f_lambda.call
# ArgumentError: wrong number of arguments (given 0, expected 1)
```

Обычный `proc` позволяет вызывать себя без аргументов, подменяя их на `nil` при исполнении. Лямбды ведут себя как совсем настоящие функции и такого не позволяют. Следующая особенность связана с обработкой оператора `return`. Посмотрите на пример:

```
f_lambda = -> { puts "in"; return } # альтернативный синтаксис для лямбд
f_proc = proc { puts "in"; return }

def fun_test(f)
  puts "before"
  f[]
  puts "after"
end

fun_test f_lambda
# before
# in
# after
#> nil
```

```
fun_test f_proc
# before
# in
# LocalJumpError: unexpected return
```

При использовании оператора `return`, `proc` осуществляет экстренный выход из метода, который его использует. Лямбда выходит сама, но не прерывает выполнение метода, в котором её вызвали. Таким образом, `proc` ведёт себя скорее как управляющая конструкция, блок, а лямбда — как самостоятельная функция. Это часто используется в функциональном Ruby — в повседневной жизни и в библиотеках вы встретите в основном их, а не `proc`'и. Блоки, лямбды и `proc`'и — примеры анонимных функций в Ruby.

Теперь посмотрим на класс `Method`. Время от времени возникает необходимость использовать в качестве анонимной функции метод какого-либо объекта. Например, нужно передать в объект, выполняющий сложную прикладную логику, метод другого объекта. На этот случай можно воспользоваться объектами класса `Method`:

```ruby
class Validators
  attr_reader :msg_prefix

  def initialize
    @msg_prefix = "Multiple criteria validator:"
  end

  def positive_validator(value)
    return true if value > 0

    puts "#{msg_prefix} the value isn't positive"
    false
  end

  # ... другие валидаторы

  def validator(name)
    method(:"#{name}_validator")
  end
end

class Validating
  attr_reader :validator

  def initialize(validator)
    @validator = validator
```

```
  end

  def validate(value)
    validator[value]
  end
end

is_positive = Validators.new.validator(:positive)
#> #<Method: Validators#positive_validator(value) (pry):88>

Validating.new(is_positive).validate(-4)
# Multiple criteria validator: the value isn't positive
#> false
```

Главная особенность Method-объектов в том, что они «помнят» контекст своего родного объекта, будучи отделёнными от него в виде анонимных функций. Эта магия называется «замыканием» и также входит в состав функциональных примитивов Ruby.

Замыкания

Полезное свойство лямбд и `proc`'ов — это лексический захват, который возникает, когда анонимные функции определяются внутри других функций. Такой приём используется при программировании, например, генераторов и называется замыканием или `closure`. Такое поведение возникает из-за того, что область видимости переменных внешней функции видима и захватывается, или замыкается внутренней. С точки зрения семантики языка, замыкание — это функция, у которой в качестве дополнительного параметра присутствует лексическое окружение. Посмотрите на пример генератора на замыкании:

```
def multiple_of(m)
  ->(n) { n * m }      # лямбда видит m и захватывает её
end

mul_3 = multiple_of 3 # это генератор

(1..4).each { |x| puts mul_3[x] }
# 3
# 6
# 9
# 12
```

Замыкания широко используются как в базовых и стандартных библиотеках Ruby, так и во фреймворке Ruby on Rails.

Коротко про пользовательские функциональные объекты. В Ruby ничто не мешает нам написать класс, единственным публичным методом которого будет `call`, а сам класс будет реализовывать сложную внутреннюю логику. Объекты, инстанциированные из такого класса, будут называться функциональными. Вот пример класса функционального объекта из реального приложения:

```
class ClearTokens < BaseInteractor
  include Dry::Monads[:result]

  TOKENS = %i[evrone_access_token evrone_refresh_token].freeze

  param :cookies

  def call
    Success(
      TOKENS.each { |tok| cookies.delete tok }
    )
  end
end
```

Карринг

Карринг базируется на довольно массивном математико-логическом фундаменте. В его основе — представление о том, что функция нескольких аргументов всегда может быть запрограммирована, как комбинация нужного количества функций на одном аргументе. Некоторые функциональные языки программирования целиком построены на этом представлении. На практике карринг означает, что из функции с некоторым количеством аргументов можно сделать функцию с меньшим количеством аргументов, абстрагируя то действие, которое совершается с «базовыми» аргументами. Далее будет пример вычисления, запрограммированного с каррингом.

Пусть в некотором алгоритме нужно часто выполнять умножение с некоторым количеством дополнительных действий и с журналированием (например, это бухгалтерские вычисления):

```
def heavy_multiply(a, b)
  puts "Осторожно умножаем #{a} на #{b}"
  # тысяча вспомогательных операций
```

```
  a * b
end
```

Через некоторое время обнаруживается, что самая частая операция в алгоритме — это умножение на 100_500, причём с теми же формальностями. Настало время сделать функцию, которая абстрагирует действие умножения на 100_500. Карринг позволяет породить такую функцию, частично применяя первоначальный метод к меньшему числу аргументов:

```
heavy_multiply_by_100500 = method(:heavy_multiply).curry[100_500]
#> => #<Proc:0x00007f9020582c50 (lambda)>

heavy_multiply(100500, 85)
# Осторожно умножаем 100500 на 85
#> 8542500

heavy_multiply_by_100500[85]
# Осторожно умножаем 100500 на 85
#> 8542500
```

Это, конечно, изолированный учебный пример. Поэтому посмотрите real-life кейс, в котором достаточно сложная логика уместилась в две строки — всё за счёт карринга и использования лямбд:

```
def call_interactor( # rubocop:disable Metrics/ParameterLists
  interactor,
  context,
  status:       :ok,
  error_status: :bad_request,
  on_success:   method(:render_jsonapi),
  on_error:      method(:render_error)
)
  interactor.call(**context)
            .either(on_success.curry[status], on_error.curry[error_status])
end
```

Композиция функций

Мы собираем нескольких функций, складывая их имена при помощи специального бинарного оператора. Смотрите, например, мы определили три функции:

```
increment = ->(x) { x + 1 }
decrement = ->(x) { x - 1 }
double    = ->(x) { x * 2 }
```

Затем нам нужно последовательно вызвать их, чтобы каждая получала в качестве аргумента результат выполнения предыдущей:

```
increment[increment[double[decrement[5]]]]
#> 10
```

Получается довольно громоздкая запись. Её можно превратить в более изящную, используя композицию функций:

```
chain_computation = decrement >> double >> increment >> increment
#> #<Proc:0x00007f90202f84f8 (lambda)>

chain_computation[5]
#> 10
```

Для функциональных объектов определены операторы композиции `<<` и `>>`. Направление шеврона указывает порядок производимых операций.

Мы разобрали базовые инструменты Ruby, обеспечивающие программирование в функциональном стиле. Эти инструменты в связке с ООП-ядром позволяют писать лаконичный, легко поддерживаемый, логически связный и надёжный код в сложных приложениях.

Понимаем, тема сложная, поэтому пишите нам свои вопросы в комментариях, мы будем рады помочь.

Теги: курсы программирования, ruby, ruby on rails, обучение ruby

Хабы: Блог компании Evrone, Ruby, Ruby on Rails

Evrone

Компания

Подписаться

Сайт Сайт Сайт Сайт

24 Карма 0 Рейтинг

Evrone @Evrone

Пользователь

Подписаться

Комментировать

Публикации

ЛУЧШИЕ ЗА СУТКИ ПОХОЖИЕ

mr-pickles
18 часов назад

16-, 8- и 4-битные форматы чисел с плавающей запятой

Средний 15 мин 6.8K

+68 89 28

ru_vds
15 часов назад

Искусство создания понятных графиков

Средний 7 мин 3.6K

Туториал Перевод

+38 79 2

dlinyj
20 часов назад

Измерение скорости чтения-записи носителей с помощью утилиты dd

Средний 11 мин 4K

Кейс

+38 48 29

SLY_G
10 часов назад

Известная, но очень странная кошачья повадка: кошки, приносящие игрушку — это эволюционная загадка

5 мин 7.8K

Перевод

+30 22 40

alizar
19 часов назад

Автономия разработчиков. Как устроены компании нового типа

Простой 7 мин 4K

Мнение

+25 20 21

Petr0v1
14 часов назад

Марс всё ближе: несмотря на проблемы, запуск Starship можно считать успешным

4 мин 4K

+16 10 12

ne_volkov
19 часов назад

Как в Ozon следят за чувствительной информацией в логах и при чем тут Толкиен?

Простой 10 мин 3.2K

Кейс

+15 22 3

Doctor_IT
15 часов назад

Сможет ли високосная минута решить проблему синхронизации часов?

6 мин 2.8K

+14 15 14

badcasedaily1
12 часов назад

Garbage Collection и JVM

Простой 17 мин 2.2K

Обзор

+13 68 3

SLY_G
16 часов назад

Дайджест научпоп-новостей за неделю, о которых мы ничего не писали

8 мин 1.6K

Дайджест

+13 5 1

Показать еще

ИНФОРМАЦИЯ

Сайт	evrone.ru
Дата регистрации	2 августа 2022
Дата основания	2008
Численность	101–200 человек
Местоположение	Россия

БЛОГ НА ХАБРЕ

19 мая в 19:04

Курс по Ruby+Rails. Часть 8. Модели и первые шаги

1.8K 0

25 апр в 14:29

Что нового в Proxmox 7.4

6.9K 23

6 апр в 14:00

Как добавить сторонние драйверы в установочный образ VMware ESXi 8

4.9K 18

22 мар в 19:40

Курс по Ruby+Rails. Часть 7. Модели и ActiveRecord

2.6K 1

27 фев в 19:55

Подробный гайд по Docker на M1

13K 6

Настройка языка

Техническая поддержка