

МГТУ им. Н.Э. Баумана

Реферат
по курсу «Парадигмы и конструкции языков программирования»
Тема: Язык программирования Crystal. История создания, фишки и плюсы
этого языка.

Проверил:
Нардид А.Н.

Подготовила:
Студент группы ИУ5-36Б
Мохаммед М. Н.

2024 г.

Зарождение языка

История Crystal начинается в 2011 году, когда команда энтузиастов решили создать язык, который бы исправил некоторые из тех ограничений и проблем, с которыми они сталкивались, работая с Ruby. Они мечтали о языке, который бы позволял писать код, легкий для понимания и поддержки, но при этом обладающий высокой производительностью и эффективностью выполнения. Так родился Crystal, язык, который наследует синтаксис Ruby.

На первый взгляд, код на Crystal может показаться почти идентичным коду на Ruby — это было сделано намеренно, чтобы разработчики, уже знакомые с Ruby, могли без труда перейти на использование нового языка. Однако, несмотря на внешнее сходство, Crystal вносит ряд улучшений: система статической типизации с автоматическим выводом типов, обработка параллельных вычислений и возможность компиляции в машинный код.

Будем рассматривать язык Crystal на конкретных примерах и в процессе также будем сравнивать синтаксис этого языка с языком Ruby.

Основной синтаксис

Crystal автоматически выводит типы переменных. Несмотря на это, можно явно указывать типы:

```
name : String = "name"  
age : Int32 = 10
```

Crystal поддерживает стандартные управляющие структуры, такие как `if`, `else`, `case`, а также циклы `while` и `until`:

```
if age >= 18  
  puts "Adult"  
else  
  puts "Minor"  
end
```

Run #hih1

Code



```
1 name : String = "name"  
2 age : Int32 = 10  
3  
4 if age >= 18  
5   puts "Adult"  
6 else  
7   puts "Minor"  
8 end
```

Sun Dec 15 2024 15:14:18 GMT+0300 (Москва, стандартное время)

Exited with: 0 | Crystal 1.14.0

Output

Minor

Определение методов в Crystal очень похоже на Ruby, но с добавлением типов аргументов и возвращаемого значения:

```
def add(a : Int32, b : Int32) : Int32
  a + b
end
```

Run #hih4

Code



```
1 def add(a : Int32, b : Int32) : Int32
2   a + b
3 end
4 puts(add(10, 20))
```

Sun Dec 15 2024 15:17:50 GMT+0300 (Москва, стандартное время)

Exited with: 0 | Crystal 1.14.0

Output

30

Ruby:

```
def add(a, b)
  a + b
end
```

```
result = add(10, 20)
puts result # Output: 30
```

ООП в Crystal реализовано через классы и модули, подобно Ruby, но с более строгой системой типов:

```
class Person
  property name : String
  property age : Int32

  def initialize(@name : String, @age : Int32)
    end
end
```

Ruby:

```
class Person #создаем класс, присваиваем ему имя Person
  attr_accessor :name, :age # аксессоры
```

```

def initialize (name, age) # конструктор класса.
  @name = name # переменная объекта
  @age = age
end
end

```

Макросы в Crystal позволяют генерировать код во время компиляции:

```

macro tag(name, content)
  {% puts "<#{name}>#{content}</#{name}>" %}
end

```

```
tag("p", "Hello, World!")
```

Run #hih6

Code



```

1 macro tag(name, content)
2   {% puts "<#{name}>#{content}</#{name}>" %}
3 end
4
5 tag("p", "Hello, World!")

```

Sun Dec 15 2024 16:20:04 GMT+0300 (Москва, стандартное время)

Exited with: 0 | Crystal 1.14.0

Output

```
<"p">"Hello, World!"</"p">
```

В Ruby нет макросов в привычном понимании

Crystal обрабатывает ошибки через систему исключений, аналогичную Ruby, но требует явного указания возможных типов исключений:

```

begin
  # опасный код
rescue ex : DivisionByZeroError
  puts "Cannot divide by zero!"
end

```

Crystal может легко взаимодействовать с C:

```

@[Link(ldflags: "-lsqlite3")]
lib LibSQLite3
  fun open(filename : String, out db : SQLite3) : Int32
end

```

Типизация

Статическая типизация означает, что тип каждой переменной, параметра и метода известен на этапе компиляции. Статическая типизация позволяет Crystal предотвращать целый ряд ошибок еще до запуска программы.

Однако статическая типизация часто ассоциируется с необходимостью явного указания типов, и все это фиксируется с помощью инференции типов — способности компилятора автоматически определять типы на основе контекста использования переменных и выражений.

Рассмотрим простой пример:

```
def add(a, b)
  a + b
end

puts add(1, 2) # 3
```

Здесь не указаны типы для параметров `a` и `b`, но благодаря инференции типов Crystal понимает, что оба параметра и результат их сложения должны быть целыми числами, по умолчанию в основном `Int32`.

Run #hih9

Code



```
1 def add(a, b)
2   a + b
3 end
4
5 puts add(1, 2) # 3
```

Sun Dec 15 2024 16:25:03 GMT+0300 (Москва, стандартное время)

Exited with: 0 | Crystal 1.14.0

Output

```
3
```

Если попытаться сложить число и строку:

```
puts add(1, "two")
```

Code



```
1 def add(a, b)
2   a + b
3 end
4
5 puts add(1, "two")
```

Sun Dec 15 2024 16:26:16 GMT+0300 (Москва, стандартное время)

Exited with: 1 | Crystal 1.14.0

Error output

Showing last frame. Use --error-trace for full trace.

error in line 2

Error: expected argument #1 to 'Int32#+' to be Float32, Float64, Int128, Int16, I

Overloads are:

- Int32#+(other : Int8)
- Int32#+(other : Int16)
- Int32#+(other : Int32)
- Int32#+(other : Int64)
- Int32#+(other : Int128)
- Int32#+(other : UInt8)
- Int32#+(other : UInt16)
- Int32#+(other : UInt32)
- Int32#+(other : UInt64)
- Int32#+(other : UInt128)
- Int32#+(other : Float32)
- Int32#+(other : Float64)
- Number#+()

Компилятор Crystal выдаст ошибку, поскольку не сможет найти подходящую перегрузку метода `add`, которая бы соответствовала таким типам аргументов.

Crystal также поддерживает универсальные типы:

```
array = [1, 2, 3] # автоматом становится Array(Int32)
```

Конкуренция и параллелизм

Модель акторов в Crystal — это абстракция, которая позволяет рассматривать каждую единицу параллельного выполнения как актора, способного обрабатывать сообщения, выполнять задачи и взаимодействовать с другими акторами, все это дает высокий уровень изоляции между акторами

Акторы могут:

- Создавать других акторов.
- Отправлять сообщения другим акторам.
- Обрабатывать входящие сообщения.

В Crystal модель акторов реализована через использование волокон и каналов.

Волокна представляют собой легковесные потоки выполнения, которые позволяют выполнять множество задач параллельно внутри одного операционного потока. В отличие от традиционных потоков, переключение контекста между волокнами происходит быстрее, поскольку оно управляется самим языком, а не ОС.

Каналы в Crystal типизированы, что означает, что канал для передачи сообщений определенного типа может передавать только сообщения этого типа.

Создадим несколько волокон для параллельной обработки данных, передаваемых через канал:

```
channel = Channel(Int32).new

# создаем волокна-производители
5.times do |producer_number|
  spawn do
    5.times do |i|
      value = producer_number * 10 + i
      puts "Producer #{producer_number} sending: #{value}"
      channel.send(value)
      sleep rand(0.1..0.5) # имитация задержки
    end
  end
end

# создаем волокно-потребитель
spawn do
  25.times do
    received = channel.receive
    puts "Consumer received: #{received}"
  end
end

sleep 3 # даем время для выполнения волокон
```

Output

```
Producer 0 sending: 0
Producer 1 sending: 10
Producer 2 sending: 20
Producer 3 sending: 30
Producer 4 sending: 40
Consumer received: 0
Consumer received: 10
Consumer received: 20
Consumer received: 30
Consumer received: 40
Producer 2 sending: 21
Consumer received: 21
Producer 3 sending: 31
Consumer received: 31
Producer 0 sending: 1
Consumer received: 1
Producer 1 sending: 11
Consumer received: 11
Producer 3 sending: 32
Consumer received: 32
Producer 4 sending: 41
Consumer received: 41
Producer 2 sending: 22
Consumer received: 22
Producer 0 sending: 2
Consumer received: 2
Producer 3 sending: 33
Consumer received: 33
Producer 1 sending: 12
```

В следующем примере юзаем каналы для сигнализации о завершении асинхронных задач:

```
done_channel = Channel(Nil).new

# асинхронная задача 1
spawn do
  sleep 1 # имитация длительной операции
  puts "Task 1 completed"
  done_channel.send(nil) # отправляем сигнал о завершении
end

# асинхронная задача 2
spawn do
  sleep 2 # имитация еще более длительной операции
  puts "Task 2 completed"
  done_channel.send(nil) # отправляем сигнал о завершении
end

2.times { done_channel.receive } # ожидаем сигналов о завершении обеих задач
puts "All tasks completed"
```


Главное волокно ожидает два сигнала о завершении, прежде чем выводить сообщение о том, что все задачи выполнены.

Полезные библиотеки

Установка библиотек в Crystal осуществляется через систему управления зависимостями под названием *Shards*. Shards аналогичен Bundler в Ruby, npm в Node.js или pip в Python и используется для управления библиотеками, на которых зависит проект.

Каждый проект на Crystal, использующий внешние зависимости, должен иметь файл конфигурации shard.yml в корневой директории проекта. Этот файл содержит метаданные проекта и список зависимостей.

После настройки файла shard.yml в терминале юзается команда shards install.

Команда скачает и установит все указанные в файле shard.yml зависимости в папку lib/ проекта. Shards также создаст файл shard.lock, который содержит версии всех установленных зависимостей

Библиотеки импортируются с помощью require.

Amber предлагает MVC архитектуру, ORM, систему шаблонов, веб-сокеты и многое другое, к примеру:

```
require "amber"

class WelcomeController < Amber::Controller::Base
  def index
    render("index.ecr")
  end
end

Amber::Server.configure do |app|
  pipeline :web do
    plug Amber::Pipe::Logger.new
    plug Amber::Pipe::Session.new
  end

  routes :web do
    get "/", WelcomeController, :index
  end
end

Amber::Server.start
```

Async позволяет легко создавать асинхронные задачи и управлять ими:

```
require "async"
```

```
Async do
  # асинхронная задача
  sleep 1
  puts "Hello from Async!"
end

puts "Hello from Main Thread!"
```

Метопрограммирование

Метапрограммирование позволяет программам генерировать и трансформировать код во время компиляции.

Допустим, есть класс, и нужно автоматически сгенерировать геттеры и сеттеры для его свойств. Это можно сделать с помощью макросов, которые упоминались выше.

Вывод:

Crystal хорош как язык программирования по следующим причинам:

- **Статическая типизация.** Ошибки несоответствия типов переменных выявляются компилятором уже на стадии переработки исходного кода в машинный, а не в процессе его выполнения интерпретатором.
- **Независимая от ОС реализация многопоточности.** Легковесные потоки в Crystal называются «волоконками» (fibers). Потоки могут взаимодействовать друг с другом посредством каналов, без необходимости прибегать к использованию общей памяти либо блокировкам.
- **Интерфейс вызова функций из библиотек языка C.** При этом синтаксис взаимодействия простой — соответственно, с использованием Crystal можно создавать библиотеки-обёртки, без необходимости писать код с нуля.
- **Широкий спектр типовых функций.** Стандартная библиотека языка представляет средства для обработки CSV, YAML, и JSON, компоненты для создания HTTP-серверов и поддержки WebSocket.
- **Высокая производительность.** По словам разработчиков, производительность приложений, написанных на Crystal, сравнима с приложениями на C.