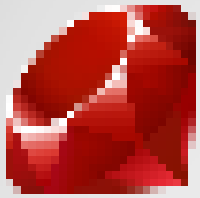

Ruby

Галаков Юрий, ВМИ-313

Человек создан для творчества, и я всегда знал, что люблю творить. Увы, я обделён талантом художника или музыканта. Зато умею писать программы. Я хочу, чтобы компьютер был моим слугой, а не господином, поэтому должен уметь быстро и эффективно объяснить ему, что делать.

Юкихиро Мацумото, создатель языка Ruby

Ruby:

- Динамический
- Рефлективный
- Интерпретируемый
- Высокоуровневый

- Ruby был задуман в 1993-м году
- Создатель - Юкихиро Мацумото
- Стал популярным с момента появления первой общедоступной версии в 1995 году
- Настоящий скачок в развитии и интерес к использованию языка для серьёзных коммерческих проектов начался после выхода Ruby 1.8.0 в 2003-м и новой версии framework'a Ruby on Rails 2.0 в 2007-м году

Основные свойства Ruby:

- Интерпретируемый язык:
  - Возможность прямых системных вызовов.
  - Мощная поддержка операций со строками и правилами (регулярными выражениями).
  - Мгновенное проявление изменений во время разработки.
  - Отсутствие стадии компиляции.
- Простое и быстрое программирование:
  - Не надо объявлять переменные.
  - Переменные динамически типизированы.
  - Простой и последовательный синтаксис.
  - Автоматическое управление оперативной памятью.
- Объектно-ориентированное программирование
  - Всё есть объект. Даже имя класса есть экземпляр класса Class.
  - Присутствуют классы, методы, наследование, полиморфизм, инкапсуляция и так далее.
  - Методы-одиночки.
  - Возможность расширить класс без наследования.
  - Итераторы и замыкания.
  - Широкие возможности метапрограммирования.

Синтаксис языка.

Синтаксис.

Синтаксический анализатор Ruby сложен и склонен прощать многие огрехи. Он пытается понять, что хотел сказать программист, а не навязывать ему жесткие правила. Но к такому поведению надо еще привыкнуть. Вот перечень того, что следует знать о синтаксисе Ruby.

Регулярные выражения в Ruby.

/Ruby/	Соответствует одному слову Ruby
/[Rr]uby/	Соответствует Ruby или ruby
/^abc/	Соответствует abc в начале строки
%r(xyz$)	Соответствует xyz в конце строки
%r|[0-9]*|	Соответствует любой последовательности из нуля или более цифр

^	Начало строки текста (line) или строки символов (string)
$	Конец строки текста или строки символов
.	Любой символ, кроме символа новой строки (если не установлен многострочный режим)
\w	Символ - часть слова (цифра, буква или знак подчеркивания)
\W	Символ, не являющийся частью слова
\s	Пропуск (пробел, знак табуляции, символ новой строки и т.д.)
\S	Символ, не являющийся пропуском
\d	Цифра (то же, что [0-9])
\D	Не цифра
\A	Начало строки символов (string)
\Z	Конец строки символов или позиция перед конечным символом новой строки
\z	Конец строки символов (string)
\b	Граница слова (только вне квадратных скобок [ ])

[]	Произвольный набор символов
*	0 или более повторений предыдущего подвыражения
*?	0 или более повторений предыдущего подвыражения (нежадный алгоритм)
+	1 или более повторений предыдущего подвыражения
+?	1 или более повторений предыдущего подвыражения (нежадный алгоритм)
{m, n}	От m до n вхождений предыдущего подвыражения
{m, n}?	От m до n вхождений предыдущего подвыражения (нежадный алгоритм)
?	0 или 1 повторений предыдущего подвыражения
|	Альтернативы
(?=)	Позитивное заглядывание вперед
(?!)	Негативное заглядывание вперед
()	Группировка подвыражений
(?>)	Вложенное подвыражение
(?:)	Несохраняющая группировка подвыражений
(?imx-imx)	Включить/выключить режимы, начиная с этого места
(?imx-imx: expr)	Включить/выключить режимы для этого выражения
(?#)	Комментарий

- Скобки при вызове методов, как правило, можно опускать.

- Коль скоро скобки необязательны, что означает такая запись:

```
1. x y z
```

Оказывается, вот что: «Вызвать метод y, передав ему параметр z, а результат передать в виде параметра методу x.» Иными словами,

```
1. x(y(z))
```

- Попробуем передать методу хэш:

```
1. my_method {a=>1, b=>2}
```

Это приведет к синтаксической ошибке, поскольку левая фигурная скобка воспринимается как начало блока. В данном случае скобки необходимы:

```
1. my_method({a=>1, b=>2})
```

- Предположим теперь, что хэш — единственный (или последний) параметр метода. Ruby снисходительно разрешает опускать фигурные скобки:

```
1. my_method(a=>1, b=>2)
```

- Есть и другие случаи, когда пропуски имеют некоторое значение. Например, на первый взгляд все четыре выражения ниже означают одно и то же:

```
1.  x = y + z
2.  x = y+z
3.  x = y+ z
4.  x = y +z
```

Но фактически эквивалентны лишь первые три. В четвертом же случае анализатор считает, что вызван метод y с параметром +z и выдаст сообщение об ошибке, так как метода с именем y не существует.

Мораль: пользуйтесь пробелами разумно.

- В именах идентификаторов знак подчеркивания _ считается строчной буквой. Следовательно, имя идентификатора может начинаться с этого знака, но такой идентификатор не будет считаться константой, даже если следующая буква прописная.

- В линейной последовательности вложенных предложений if применяется ключевое слово *elsif*, а не *else if* или *elif*, как в некоторых других языках.

- Ключевые слова в Ruby нельзя назвать по-настоящему зарезервированными. Если метод вызывается от имени некоторого объекта (и в других случаях, когда не возникает неоднозначности), имя метода может совпадать с ключевым словом. Но поступайте так с осторожностью, не забывая, что программу будут читать люди.

- *then*(в предложениях *if* и *case*) необязательно
- То же относится к слову *do* в циклах *while* и *until*.
- Вопросительный и восклицательный знаки не являются частью идентификатора, который модифицируют(*chop* и *chop!)*
- Аналогично в Ruby есть конструкция *defined?*, но *defined*— ключевое слово.

Переменные.

- Переменные используются, чтобы сохранить промежуточный результат вычислений. Имя переменной в Ruby должно:
  a) начинаться с буквы или знака подчёркивания;
  b) состоять из латинских букв, цифр и знака подчёркивания.
- имена глобальных переменных начинаются со знака $
- имена переменных экземпляра (принадлежащих объекту) начинаются со знака @
- имена переменных класса (принадлежащих классу) предваряются двумя знаками @@

- Переменные указывают на объект, т.е. если изменяется значение объекта, то изменятся значения ВСЕХ переменных, на него указывающих.

Пример:

```
1. girlfriend = "Даша"
2. babe = girlfriend
3. puts girlfriend
4. =>Даша
5. babe[0] = "М"
6. puts girlfriend
7. =>Маша
```

- В большинстве языков для обмена значений двух переменных нужна дополнительная временная переменная. В Ruby наличие механизма множественного присваивания делает ее излишней: выражение

```
1. x, y = y, x
```

обменивает значения x и y местами.

Классы.

Классы.

- В Ruby есть множество встроенных классов, и вы сами можете определять новые. Для определения нового класса применяется такая конструкция:

```
1. class ClassName
2. # ...
3. end
```

- Само имя класса - это глобальная константа, поэтому оно должно начинаться с прописной буквы. Определение класса может содержать константы, переменные класса, методы класса, переменные экземпляра и методы экземпляра. Данные уровня класса доступны всем объектам этого класса, тогда как данные уровня экземпляра доступны только одному объекту.

Методы.

Методы.

- Методы в *Ruby* обычно используются в сочетании с простыми экземплярами классов и переменными, причем вызывающий объект отделяется от имени метода точкой:

```
1.  receiver.method
```

- Если имя метода является знаком препинания, то точка опускается.
- У методов могут быть аргументы:

```
1.  Time.mktime(2000, "Aug", 24, 16, 0)
```

- Методы могут принимать переменное число аргументов:

```
1.  receiver.method(arg1, *more_args)
```

В данном случае вызванный метод трактует *more_args* как массив и обращается с ним, как с любым другим массивом.

- Так же, методы могут «сцепляться»:

```
1.  3.succ.to_s
```

Но при этом нужно помнить, что при определенных условиях некоторые методы возвращают *nil*, а вызов любого метода от имени такого объекта приведет к ошибке.

method_missing

Пример 1.

```
1. Post.find_by_title("Awesomeness!")
2. User.find_by_email("bob@example.com")
3. User.find_by_email_and_login("bob@example.com", "bob")
```

Пример 2.

```
1. class ActiveRecord::Base
2.   def method_missing(meth, *args, &block)
3.     if meth.to_s =~ /^find_by_(.+)$/
4.       run_find_by_method($1, *args, &block)
5.     else
6.       super # Вы должны вызвать super если вы не собираетесь
7.       # использовать метод, иначе вы засорите поиск методов.
8.     end
9.   end
```

Наследование. Инкапсуляция. Полиморфизм.

Наследование.

- Ruby поддерживает только одиночное наследование, и все классы образуют единую иерархию с одним общим корнем — классом *Object*. Базовый класс указывается при помощи специального синтаксиса во время описания класса:

```
1. class Derived < Base
2. ...
3. end
```

- Определить суперкласс любого класса можно с помощью метода *superclass*
- Если во время описания явно не задать имя суперкласса, то Ruby автоматически использует в качестве суперкласса *Object*

- Если производный класс определяет собственный конструктор, то конструктор базового типа автоматически не вызывается, это нужно делать явно с помощью ключевого слова super:

```
1. class Derived < Base
2.     def initialize
3.      puts "Derived#initialize”
4.      super
5.     end
6. end
```

- В Ruby все методы класса являются, в терминологии других языков, виртуальными. Т.е. любой метод можно переопределить в производном классе. А т.к. в Ruby нет возможности указать, что какой-то конкретный метод больше не может быть переопределен, то если в конструкторе суперкласса вызывается метод, перекрытый в производном классе, то вызывается именно перекрытая производным классом версия метода.

Инкапсуляция.

Область применения метода объявляется с помощью частных методов экземпляров из класса *Module*: .*public*, .*private* и .*protected*. Ограничивается применение указанных методов или методов, объявляемых после. Методы, определяемые вне тела класса, относятся к частным методам класса *Object*.

```
1.  .private_class_method(*name) # -> self [Module]
```

Используется для объявления методов класса частными. Обычно применяется для инкапсуляции конструкторов.

```
1.  .public_class_method(*name) # -> self [Module]
```

Используется для объявления методов класса общими.

```
1.  .module_function(*name) # -> self [PRIVATE: Module]
```

Используется для объявления методов экземпляров частными и создания аналогичных методов модуля.

Полиморфизм.

Ruby поддерживает интерфейсный полиморфизм. Он позволяет определять модули, методы которых допускается «подмешивать» к существующим классам. Но обычно модули так не используются. Модуль состоит из методов и констант, которые можно использовать так, будто они являются частью класса или объекта.

Когда модуль подмешивается с помощью предложения include, мы получаем ограниченную форму множественного наследования. (По словам проектировщика языка Юкихиро Мацумото, это можно рассматривать как одиночное наследование с разделением реализации.) Таким образом удается сохранить преимущества множественного наследования, не страдая от его недостатков.

Передача сообщений.

Передача сообщений.

На самом деле вызов метода - это передача (send) ему сообщения.

Наглядно:

```
1. # Это
2. 1 + 2 # то же самое, что и ...
3. 1.+(2) # и то же самое, что и:
4. 1.send "+", 2
```

Еще немного о передаче сообщений:

Синтаксический сахар	Без сахара	Явный вызов send
10 % 3	10.modulo(3)	10.send(:modulo, 3)
5+3	5.+(3)	5.send(:+, 3)
x == y	x.==(y)	x.send(:==, y)
a * x + y	a.*(x).+(y)	a.send(:*, x).send(:+, y)
a + x * y	a.+(x.*(y))	a.send(:+, x.send(:*, y))
x[3]	x.[](3)	x.send(:[], 3)
x[3] = 'a'	x.[]=(3,'a')	x.send(:[]=, 3, 'a')
/abc/, %r{abc}	Regexp.new("abc")	Regexp.send(:new, 'abc')
str =~ regex	str.match(regex)	str.send(:match, regex)
regex =~ str	regex.match(str)	regex.send(:match, str)
$1. . . $n (regex capture)	Regexp.last_match(n)	Regexp.send(:last_match,n)

Обработка исключительных ситуаций.
Ruby поддерживает гибкую схему обработки исключений:

```
1. begin
2. #код, который может вызвать исключения
3. rescue
4. [имя классов исключений] #код, выполняемый в случае исключения
5. rescue
6. [имя классов исключений] #код, выполняемый в случае исключения
7. ensure
8. #код, который выполняется в любом случае
9. end
```

Если в секции begin происходит исключение, выполняется секция с соответствующим именем исключения.
Секция ensure выполняется в любом случае.
Секции rescue и ensure могут быть опущены. Если в rescue не указан класс исключения, подразумевается исключение StandardError, и rescue выполняется для тех исключений, которые находятся в отношении is_a? по отношению к StandardError, то есть сам StandardError и его наследники. Все выражение в целом возвращает значение секции begin. Самое последнее исключение хранится в глобальной переменной $! (и его тип определяется с помощью $!.type).

Пример программы:

```
1.  print "Введите температуру и шкалу (C or F): "
2.  str = gets
3.  exit if str.nil? or str.empty?
4.  str.chomp!
5.  temp, scale = str.split(" ")
6.  abort "#{temp} недопустимое число."
7.    if temp !~ /-?\d+/temp = temp.to_f case scale
8.    when " C", "c"f = 1.8*temp + 32
9.    when "F", "f"c = (5.0/9.0)*(temp-32)
10.    else abort "Необходимо задать C или F."
11.   end
12.   if f.nil?print "#{c} градусов C\n"
13.    else print "#{f} градусов F\n"
14.   end
```

Ruby on rails.

Ruby on rails.

- Ruby on Rails — фреймворк, написанный на языке программирования Ruby. Ruby on Rails предоставляет архитектурный образец Model-View-Controller (модель-представление-контроллер) для веб-приложений, а также обеспечивает их интеграцию с веб-сервером и сервером базы данных.
- Ruby on Rails является открытым программным обеспечением и распространяется под лицензией MIT.
- Ruby on Rails существенно использовался при создании таких популярных сайтов, как Твиттер, Diaspora, Look At Me, Groupon, Basecamp, GitHub, Hulu, Scribd, Shopify, Yellowpages.com, Danbooru.

Принципы

Ruby on Rails определяет следующие принципы разработки приложений:

- Предоставляет механизмы повторного использования, позволяющие минимизировать дублирование кода в приложениях (принцип Don’t repeat yourself).
- По умолчанию используются соглашения по конфигурации, типичные для большинства приложений (принцип Convention over configuration). Явная спецификация конфигурации требуется только в нестандартных случаях.

Архитектура.

Основными компонентами приложений Ruby on Rails являются модель (model), представление (view) и контроллер (controller). Ruby on Rails использует REST-стиль построения веб-приложений.

Модель.

- Модель предоставляет остальным компонентам приложения объектно-ориентированное отображение данных (таких как каталог продуктов или список заказов). Объекты модели могут осуществлять загрузку и сохранение данных в реляционной базе данных, а также реализуют бизнес-логику.
- Для хранения объектов модели в реляционной СУБД по умолчанию в Rails 3 использована библиотека ActiveRecord. Конкурирующий аналог — DataMapper. Существуют плагины для работы с нереляционными базами данных, например Mongoid для работы с MongoDB.

Представление.

- Представление создает пользовательский интерфейс с использованием полученных от контроллера данных. Представление также передает запросы пользователя на манипуляцию данными в контроллер (как правило, представление не изменяет непосредственно модель).
- В Ruby on Rails представление описывается при помощи шаблонов ERB. Они представляют собой файлы HTML с дополнительными включениями фрагментов кода Ruby (Embedded Ruby или ERb). Вывод, сгенерированный встроенным кодом Ruby, включается в текст шаблона, после чего получившаяся страница HTML возвращается пользователю. Кроме ERB возможно использовать ещё около 20 шаблонизаторов, в том числе Haml.

Контроллер.

Контроллер в Rails — это набор логики, запускаемой после получения HTTP-запроса сервером. Контроллер отвечает за вызов методов модели и запускает формирование представления.

Соответствие интернет-адреса с контроллером задается в файле config/routes.rb

Контроллером в Ruby on Rails является класс, наследованный от ActionController::Base. Открытые методы контроллера являются так называемыми действиями (actions). Action часто соответствует отдельному представлению. Например, по запросу пользователя admin/list будет вызван метод list класса AdminController и затем использовано представление list.html.erb.

«Гемы»

- Вокруг Rails сложилась большая экосистема плагинов — подключаемых «гемов» (англ. gem), некоторые из них со временем были включены в базовую поставку Rails, например Sass и CoffeeScript, другие же, хотя и не были включены в базовую поставку, являются фактическим стандартом для большинства разработчиков, например, средство модульного тестирования Rspec.
- Начиная с 3-й версии Rails наблюдается тенденция вынесения части функционала в отдельные «гемы», отчасти из-за их более быстрого развития, чем сам Rails, отчасти для облегчения фреймворка.
- Количество разработчиков, сделавших хотя бы одну модификацию в код Rails на начало 2013 года составило 2782 человека.

Спасибо за внимание!