

Язык Clojure

Небольшое введение

Alex Ott

<http://alexott.net>

MarginCon 2010

О чем пойдет речь?

- 1 Что такое Clojure?
- 2 Основы языка
- 3 Взаимодействие с Java
- 4 Конкурентное программирование
- 5 Clojure в реальной жизни
- 6 Источники информации

Что такое Clojure?

- Lisp'образный язык, созданный Rich Hickey. Анонсирован в 2007-м году.
- Спроектирован для работы на существующих платформах – JVM и .Net (в разработке)
- Функциональный язык с неизменяемыми, по умолчанию, данными
- Упор на поддержку конкурентного выполнения кода
- Открытый исходный код и либеральная лицензия



Причины создания нового языка

- Lisp, но не отягощенный совместимостью с предыдущими версиями/диалектами
- Неизменяемость данных и больший упор на ФП
- Поддержка конкурентного программирования на уровне языка
- Лучшая интеграция с целевыми платформами по сравнению с прямым переносом существующих языков



- Динамически-типизированный язык
- Простой синтаксис, как у всех Lisp'ов
- Поддержка интерактивной разработки
- Спроектирован в терминах абстракций
- Метапрограммирование
- Мультиметоды и протоколы (версия 1.2)
- Компилируется в байт-код целевых платформ
- Доступ к большому количеству существующих библиотек

Отличия от других Лиспов

- Измененный синтаксис, с меньшим кол-вом скобок
- Изменения в наименовании
- Больше first-class структур данных – отображения (map), наборы (set), вектора
- Неизменяемые, по умолчанию, данные
- Связывание мета-данных с переменными и функциями
- Ленивые коллекции
- Нет поддержки макросов для процедуры чтения
- Регистро-зависимые имена
- Отсутствие tail call optimization (ограничение JVM) – явные циклы loop/recur
- Исключения вместо сигналов и рестартов

Базовые типы данных

- Целые числа произвольного размера – 14235344564564564564
- Рациональные числа – 26/7
- вещественные числа – 1.2345 и BigDecimals – 1.2345M
- Строки (String из Java) – "hello world"
- Буквы (Character в Java) – \a, \b, ...
- Регулярные выражения – #"[abc]*"
- Boolean – true и false
- nil – также как null в Java, не является пустым списком как в Lisp
- Символы (symbol) – test, var1, ...
- Ключевые символы (keywords) – :test, :hello, ...

Структуры данных

- Отдельный синтаксис для разных коллекций:
 - Списки – (1 2 3 "abc")
 - Вектора – [1 2 3 "abc"]
 - Отображения (maps) – {:key1 1234 :key2 "value"}
 - Наборы (sets) – #{:val1 "text" 1 2 10}
- Последовательность – абстракция для работы со всеми коллекциями (в том числе и классов Java/.Net)
- Общая библиотека функций для работы с последовательностями
- Ленивые операции над последовательностями
- Вектора, отображения, наборы сами являются функциями – упрощение доступа к данным
- Стабильные (persistent) коллекции
- “Переходные” (transients) коллекции – оптимизация производительности
- defstruct – оптимизация отображений для объявления сложных структур

- Простой синтаксис – программа как структуры данных
- Процедура чтения (reader) преобразует программу в структуры данных
- Все объекты представляют сами себя за исключением символов и списков
- Символы связывают значение с “переменной”
- Списки рассматриваются как формы, которые могут быть:
 - Специальной формой – `def`, `let`, `if`, `loop`, ...
 - Макросом
 - Функцией, или выражением, которое приводится к функции (`maps`, `keywords`, ...)

Пример кода

```
(defn fibo
  ([ ] (concat [1 1] (fibo 1 1)))
  ([a b]
   (let [n (+ a b)]
     (lazy-seq (cons n (fibo b n))))))
(take 100000000 (fibo))
```

```
(defn vrange2 [n]
  (loop [i 0
        v (transient [ ])]
    (if (< i n)
      (recur (inc i) (conj! v i))
      (persistent! v))))
```

- Функции как first-class objects
- Возможность разного кол-ва аргументов в функциях
- Определение функций: `defn` – top-level функции, `fn` – анонимные функции
- Упрощенный синтаксис для анонимных функций – `#(code)`.

Например:

```
(map #(.toUpperCase %) ["test" "hello"])  
(map #(vector %1 %2) [1 2 3] [4 5 6])
```

- Возможность задания тестов и pre/post-условий в метаданных.
Например,

```
(defn constrained-sqr [x]  
  {:pre [(pos? x)] :post [(> % 16), (< % 225)]}  
  (* x x))
```

- Функция компилируется в отдельный класс Java

Метапрограммирование и макросы

- Макрос получает на вход код и возвращает новый код, который будет откомпилирован
- Очень большая часть языка написана с помощью макросов
- Также как и функции, могут иметь переменной число аргументов
- Код можно генерировать с помощью функций для работы со списками, но удобней пользоваться `quasi-quote` – ‘ и операторами подстановки – , и `~@`
- Суффикс `#` в именах используется для генерации уникальных имен
- `macroexpand-1` & `macroexpand` – средства отладки

Примеры макросов

Стандартный макрос `when`:

```
(defmacro when
  [test & body]
  (list 'if test (cons 'do body)))
```

при следующем использовании

```
(when (pos? a)
  (println "positive") (/ b a))
```

раскроется в:

```
(if (pos? a)
  (do
    (println "positive")
    (/ b a)))
```

Примеры макросов

Стандартный макрос `and`

```
(defmacro and
  ([] true)
  ([x] x)
  ([x & next]
    '(let [and# ~x]
      (if and# (and ~@next) and#))))
```

раскрывается в разные формы, в зависимости от числа аргументов:

```
user> (macroexpand '(and ))           ==> true
user> (macroexpand '(and (= 1 2)))    ==> (= 1 2)
user> (macroexpand '(and (= 1 2) (= 3 3))) ==>
(let* [and__4457__auto__ (= 1 2)]
  (if and__4457__auto__
    (clojure.core/and (= 3 3))
    and__4457__auto__))
```

- Расширяемость
- Мультиметоды не привязаны к типам/классам
- Диспатчеризация на основании нескольких параметров
- Решение принимается на основании результата функции диспатчеризации
- Отличается от CLOS – отсутствие `:before`, `:after`, ...
- `defmulti` – аналог `defgeneric`.
- Определяется как `(defmulti func-name dispatch-fn)` + набор определений конкретных методов (через `defmethod`)

Пример мультиметодов

Простой пример диспатчеризации по классу:

```
(defmulti m-example class)
(defmethod m-example String [this]
  (println "This is string '" this "'"))
(defmethod m-example java.util.Collection [this]
  (print "This is collection!"))
```

и получим при запуске:

```
(m-example "Hello") ==> "This is string 'Hello '"
(m-example [1 2 3]) ==> "This is collection!"
(m-example '(1 2 3)) ==> "This is collection!"
```


Более сложный пример диспатчеризации

```
(defmulti encounter
  (fn [x y] [(:Species x) (:Species y)]))
(defmethod encounter [:Bunny :Lion] [b l] :run-away)
(defmethod encounter [:Lion :Bunny] [l b] :eat)
(defmethod encounter [:Lion :Lion] [l1 l2] :fight)
(defmethod encounter [:Bunny :Bunny] [b1 b2] :mate)
(def b1 {:Species :Bunny})
(def b2 {:Species :Bunny})
(def l1 {:Species :Lion})
(def l2 {:Species :Lion})

(encounter b1 b2) ==> :mate
(encounter b1 l1) ==> :run-away
(encounter l1 b1) ==> :eat
(encounter l1 l2) ==> :fight
```

Протоколы и типы данных

- Введены в версии 1.2
- Быстрее чем мультиметоды
- Позволяют написать Clojure in Clojure
- Диспатчеризация только по типу данных
- Похожи на type classes в Haskell
- Создают соответствующие интерфейсы для Java
- defrecord & deftype объявляют новые типы данных
- extend-protocol & extend-type связывают протокол с типами данных (не только объявленными в Clojure!)
- reify – для реализации протоколов и интерфейсов для “одноразовых” типов

Пример протоколов

```
(defprotocol Hello "Test of protocol"  
  (hello [this] "hello function"))
```

```
(defrecord B [name] Hello  
  (hello [this] (str "Hello " (:name this) "!")))  
(hello (B. "User")) ==> "Hello User!"
```

```
(extend-protocol Hello String  
  (hello [this] (str "Hello " this "!")))  
(hello "world") ==> "Hello world!"
```

```
(extend-protocol Hello java.lang.Object  
  (hello [this] (str "Hello '" this  
                    "'! (" (type this) ")")))  
(hello 1) ==> "Hello '1'! (class java.lang.Integer)"
```

- Метаданные – `#^{}` в 1.0 и 1.1 или просто `^{}` в 1.2
 - Опциональная спецификация типов (type hints) – `#^Type` или `^Type`
 - Спецификация тестов прямо в объявлении функции
 - Управление областью видимости
 - Программный доступ к метаданным
 - Не влияют на равенство (equality) значений
- Пространства имен (namespaces)
 - First-class objects
 - Используются для организации кода в библиотеки
- Деструктуризация параметров функций и возвращаемых значений

```
(let [[a b & c :as e] [1 2 3 4]] [a b c e])  
=> [1 2 (3 4) [1 2 3 4]]
```

```
(let [{:keys [a b c] :as m :or {b 3}} {:a 5 :c 6}]  
  [a b c m]) => [5 3 6 {:a 5 :c 6}]
```

- Двухстороннее взаимодействие с целевой платформой:
 - Создание экземпляров классов Java – `new` или `Class`.
 - Вызов кода, написанного на Java – `.`, `..`, `doto`
 - Генерация классов и интерфейсов для вызова из Java – `gen-class`, `gen-interface`, `proxy` (анонимный класс)
- Возможность выполнения кода Clojure из программ на Java
- Отдельные функции для работы с массивами Java
 - `make-array` – создание массивов
 - `aget/aset` – доступ к элементам массивов
 - `amap/areduce` – итерация по элементам массивов
- `memfn` позволяет использовать функции-члены классов в `map`, `filter`, ...
- Спец. форма `set!` для установки значений в классе
- Генерация и перехват исключений – `throw & catch`

Примеры

- Создание объектов:

`(new java.util.Date) <=> (java.util.Date.)`

- Доступ к полям/функциям членам классов:

`(.substring "Hello World" 0 5) ==> "Hello"`

`(. "Hello World" substring 0 5) ==> "Hello"`

`Math/PI ==> 3.141592653589793`

`(Integer/parseInt "42") ==> 42`

- .. для СВЯЗАННЫХ ВЫЗОВОВ:

`(.. System getProperties (get "os.name")) ==> "Mac
System.getProperties().get("os.name")`

- `doto` – вызов нескольких методов для одного объекта:

`(doto (java.util.HashMap.)
 (.put "a" 1) (.put "b" 2))`

Конкурентное программирование

- Средства, обеспечивающие изменяемость данных:
 - Ссылки (refs) – синхронное, координированное изменение
 - Агенты (agents) – асинхронное, независимое
 - Атомы (atoms) – синхронное, независимое
 - Переменные (vars) – изменение, локальное для нитей
 - @ (deref) – используется для доступа к данным
- Параллельное выполнение кода:
 - futures
 - pmap & pcalls
 - Native threads
- Средства синхронизации – promise

- Координированное изменение данных из нескольких потоков
- Software Transaction Memory обеспечивает атомарность, целостность, изоляцию
- Изменения происходят только в рамках транзакции
- Возможность проверки данных с помощью функции-валидатора
- Возможность добавления функций-наблюдателей

```
(def counters (ref {}))  
(defn get-counter [key]  
  (@counters key 0))  
(defn increment-counter [key]  
  (dosync  
    (alter counters assoc key  
              (inc (@counters key 0)))))
```


- Асинхронное обновление данных – “fire and forget”
- Пулы потоков для выполнения функций обновления данных: `send` – для “быстрого” обновления данных, `send-off` – для “тяжелых” функций (отдельный пул потоков выполнения)
- `send` & `send-off` получают функцию, которая будет применена к текущему состоянию агента
- Функции-валидаторы и функции-наблюдатели
- Возможность ожидания окончания всех заданий агента

```
(def counters (agent {}))  
(defn increment-counter [key]  
  (send counters assoc key  
    (inc (@counters key 0))))  
(defn get-counter [key]  
  (@counters key 0))
```

- Атомы

- Синхронное изменение данных без координации
- Изменение данных производится функцией, которая применяется к текущему значению
- Функция может быть вызвана несколько раз, если кто-то уже изменил значение
- Функция не должна иметь побочных эффектов!

- Vars

- Обеспечивают изменение данных в рамках текущего потока
- `binding` связывает новые значения с символами
- Изменения затрагивают все вызываемые из текущего потока функции
- Будьте осторожны с ленивыми последовательностями!

```
(def *var* 5)
(defn foo [] *var*)
(foo) ==> 5
(binding [*var* 10] (foo)) ==> 10
```

Параллельное выполнение кода

- `future`

- выполняет заданный код в отдельном потоке
- `@` – для доступа к результатам выполнения кода
- `@` блокирует текущий поток, если нет результатов

- `promise`

- используется для синхронизации между потоками данных
- `deliver` устанавливает значение
- `@` читает установленное значение или блокирует выполнение, если оно не установлено

- `rpar` & `pcalls` – выгодно использовать только для “тяжелых” функций обработки данных.

Clojure в реальной жизни

- Инфраструктура и инструментальная поддержка:
 - Среда разработки
 - Средства сборки
 - Библиотеки
 - Репозитории кода
- Clojure/core – коммерческая поддержка, консультации и т.п.
- Использование в коммерческих проектах
- Источники информации

Инфраструктура: IDE и средства сборки

- Поддержка в большинстве IDE:
 - Emacs + SLIME (самый популярный)
 - VimClojure
 - NetBeans
 - Eclipse
 - IntelliJ IDEA
- Утилиты сборки кода:
 - Поддержка Clojure в Maven и Ant
 - Leiningen – написан на Clojure, расширяемый и очень простой в использовании

Инфраструктура: библиотеки и репозитории

- Доступ к большому набору существующих библиотек
- Clojure-specific libraries:
 - Clojure-Contrib
 - Compojure
 - ClojureQL
 - Incanter
 - и другие – см. <http://clojars.org>
- Репозитории:
 - build.clojure.org
 - clojars.org

- Зарубежные проекты:
 - FlightCaster
 - ThinkRelevance
 - Runa
 - Sonian Networks
 - BackType
 - DRW Trading Group
 - Snowtide Informatics Systems, Inc. - проект DocuHarvest
 - ThorTech Solutions
- Несколько российских стартапов используют Clojure:
 - ООО "Моделирование и Технологии"
 - Security Technology Research (<http://setere.com>)

- Основные:

- Сайт проекта – <http://clojure.org>
- Planet Clojure – <http://planet.clojure.in>
- Канал #clojure на freenode.net
- Проект labrepl (<http://github.com/relevance/labrepl>) – учебная среда для изучения языка
- Try-Clojure (<http://www.try-clojure.org/>) – работа с кодом используя только Web-браузер

- Русскоязычные ресурсы:

- Русская планета ФП – <http://fprog.ru/planet/>
- clojure at conference.jabber.ru
- Clojure форум на <http://lisper.ru>
- Мой сайт – <http://alexott.net/ru/clojure/>

- Книги:
 - Programming Clojure – 2009-й год, версия 1.0
 - Practical Clojure. The Definitive Guide – 2010, версия 1.2
 - The Joy of Clojure (beta)
 - Clojure in Action (beta)
 - Clojure Programming on Wikibooks
 - Clojure Notes on RubyLearning
- Видео-лекции и скринкасты
- Группы пользователей по всему миру
- Учебные курсы (пока только в США и Европе)

Вопросы

