Конкурентное и параллельное программирование в Clojure

Alex Ott, 25.01.2014

О чем пойдет речь?

- Общая картина
- Изменяемое состояние
- Конкурентное и параллельное программирование
- Advanced topics
- Ресурсы

Общая картина (State & Identity)

Часть 1:

Изменяемое состояние

Изменяемое состояние

Вид изменения	Синхронное	Асинхронное
Координированное	Ссылки (refs)	
Независимое	Атомы	Агенты
Изолированное	Vars	

`deref` или **`@`** для доступа к текущему состоянию ссылки, атома или агента

Ссылки

- Синхронное, координированное изменение данных
- Основаны на Software Transactional Memory (MVCC)
- Atomicity, Consistence, Isolation
- Изменения только в рамках транзакций!
- Транзакция повторяется при конфликте с другой транзакцией
- Транзакция прекращается при генерации исключения
- Поддержка функций-валидаторов и функций-наблюдателей

Ссылки

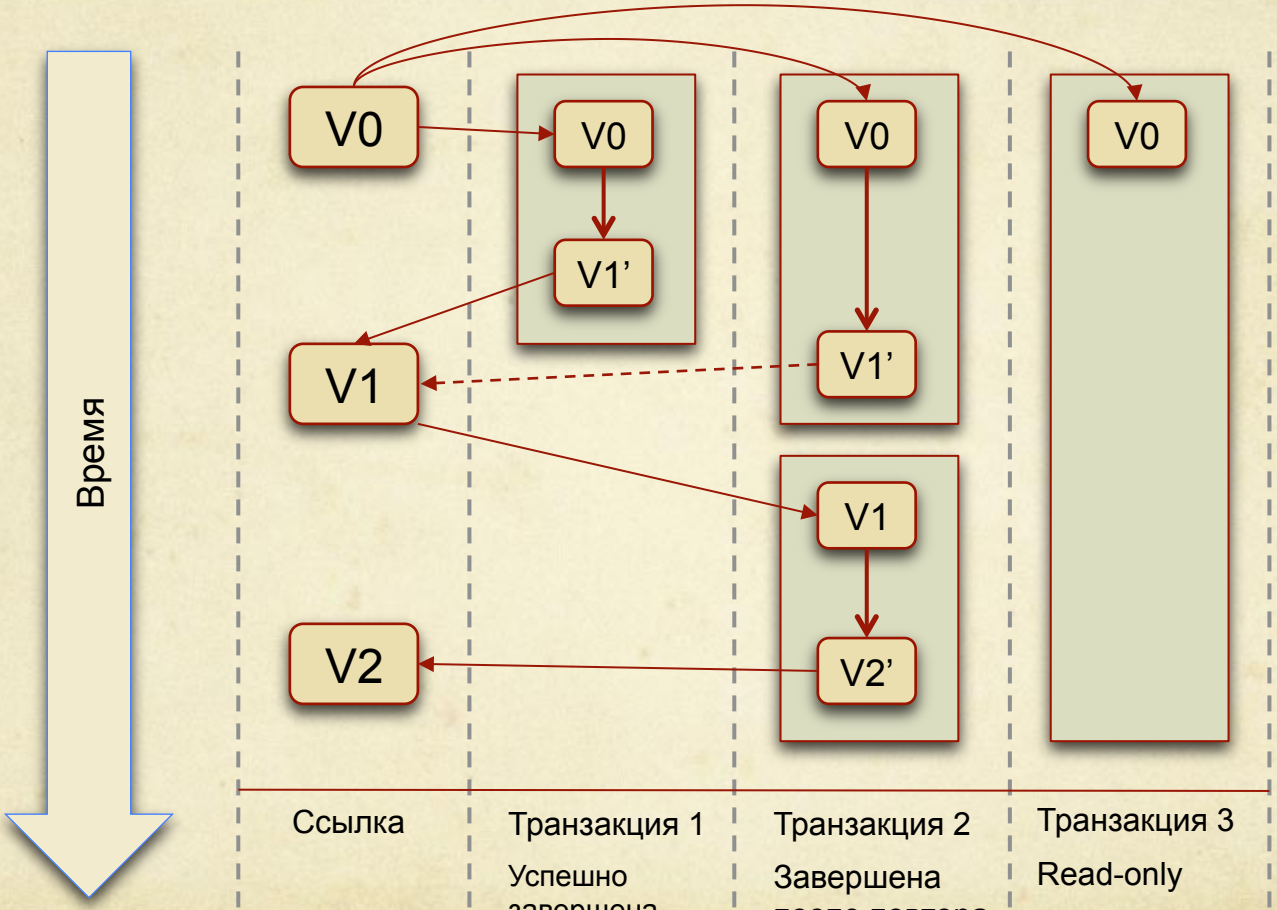


Схема взята из статьи http://sw1nn.com/blog/2012/04/11/clojure-stm-what-why-how/

Ссылки: использование

- Создание: `(ref x & опции)`
- Начало транзакции: `(dosync …)`
- Изменение: `alter`, `commute` или `ref-set`
- Блокировка: `ensure`

Ссылки: пример

```
(defn transfer-money [from to amount]
  (dosync
   (if (< @from amount)
     (throw (IllegalStateException.
         (str "Account has less money that required! "
              @from " < " amount)))
     (do (alter from - amount)
         (alter to + amount)))))
```

Ссылки: пример

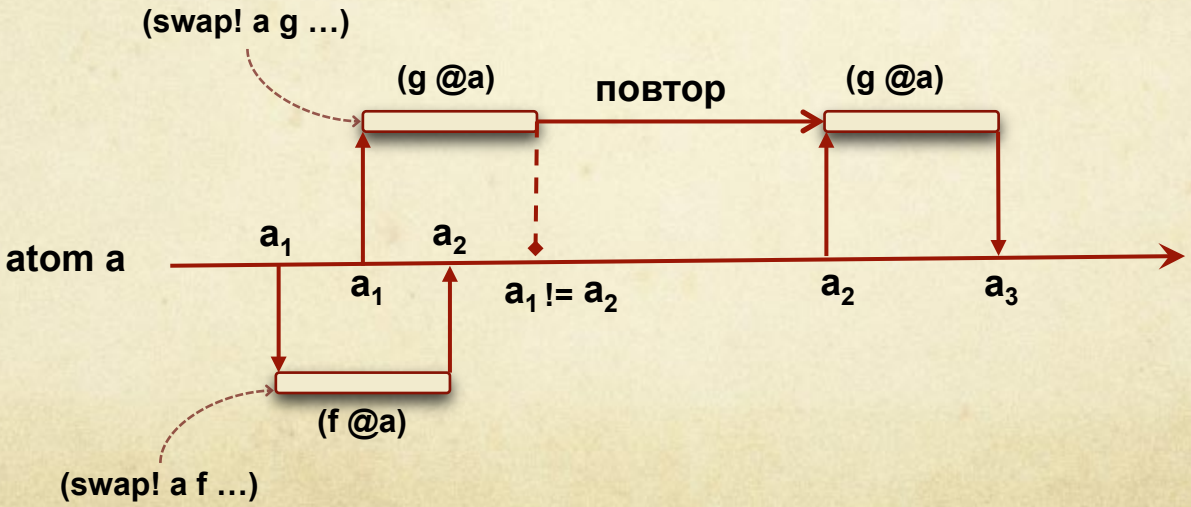
```
user=> (def ^:private acc-1 (ref 1000))
user=> (def ^:private acc-2 (ref 1000))

user=> (transfer-money acc-1 acc-2 500)
1500
user=> @acc-1
500
user=> @acc-2
1500
```

Ссылки: пример

```
(defn add-to-deposit [to amount]
  (dosync
   (commute to + amount)))

user=> (add-to-deposit acc-1 100)
600
user=> @acc-1
600

(defn write-log [log-msg]
  (io!
   (println log-msg)))
user=>(dosync (write-log "test"))
java.lang.IllegalStateException: I/O in transaction
```

Атомы

- Синхронное, некоординированное изменение
- Основная функция – **`swap!`**
- Поддержка валидаторов и наблюдателей
- Прекращает повторы при исключении



Атомы: пример

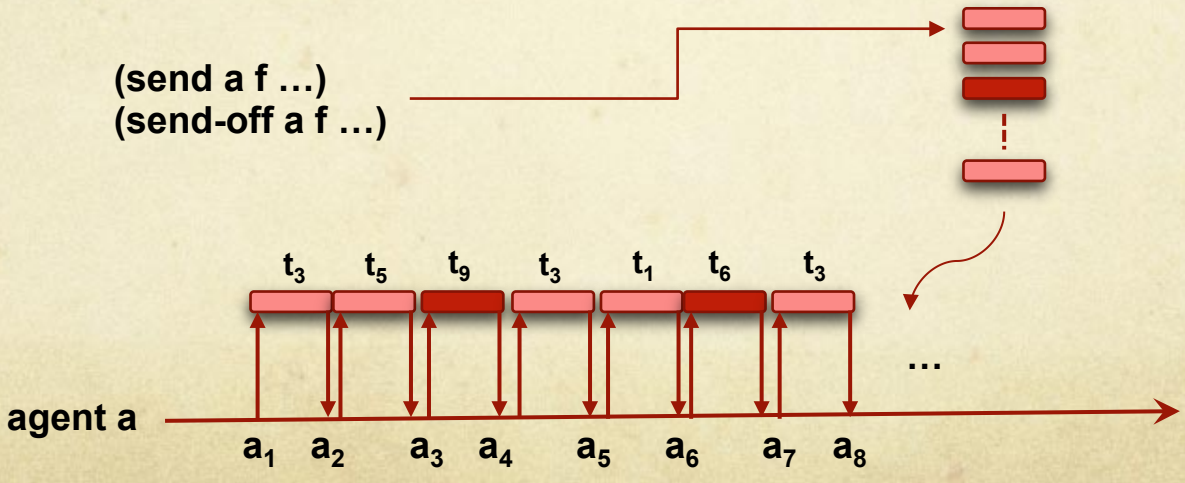
```
(def ^:private counters-atom (atom {}))

(defn inc-counter [name]
  (swap! counters-atom update-in [name]
         (fnil inc 0)))

(defn dec-counter [name]
  (swap! counters-atom update-in [name]
         (fnil dec 0)))

(defn reset-counter [name]
  (swap! counters-atom assoc name 0))
```

Атомы: пример

```
user=> @counters-atom
{}
user=> (inc-counter :test)
{:test 1}
user=> (inc-counter :another-test)
{:another-test 1, :test 1}
user=> (reset-counter :test)
{:another-test 1, :test 0}
```

Агенты

- Асинхронное, некоординированное изменение – fire & forget
- Функции: `send` – bounded thread pool, `send-off` – unbounded thread pool
- Валидаторы и наблюдатели
- Возможность обработки ошибок при выполнении кода



Агенты: пример

```
(def ^:private counters-agent (agent {}))

(defn a-inc-counter [name]
  (send counters-agent update-in [name]
        (fnil inc 0)))

(defn a-dec-counter [name]
  (send counters-agent update-in [name]
        (fnil dec 0)))

(defn a-reset-counter [name]
  (send counters-agent assoc name 0))
```

Агенты и ошибки

```
user=> (def err-agent (agent 1))
#'user/err-agent
user=> (send err-agent (fn [_] (throw
             (Exception. "we have a problem!"))))
#<Agent@8e7da60 FAILED: 1>
user=> (send err-agent identity)
Exception we have a problem!  user/eval1227/fn--1228
(form-init6590526999427540299.clj:1)

user=> (def err-agent (agent 1
            :error-mode :continue))
#'user/err-agent
user=> (send err-agent (fn [_] (throw
             (Exception. "we have a problem!"))))
#<Agent@76aa3e9a: 1>
user=> (send err-agent inc)
#<Agent@76aa3e9a: 2>
user=> @err-agent
2
```

Vars

- Изолированное изменение в рамках одного потока
- Изменение применяется ко всему вызываемому коду
- Var должна быть объявлена как `:dynamic`
- `binding` – переопределение значений:
  - Работает при использовании agents, pmap & futures
  - Не работает с lazy sequences
- `alter-var-root` – изменение top-level значения

Vars: примеры

```
(def ^:dynamic *test-var* 20)

(defn add-to-var [num]
   (+ num *test-var*)

(defn print-var [txt]
   (println txt *test-var*))

(defn run-thread [x]
  (.run (fn []
          (print-var (str "Thread " x " before:"))
          (binding [*test-var* (rand-int 10000)]
            (print-var (str "Thread " x " after:"))))))

user=> (doseq [x (range 3)] (run-thread x))
Thread 0 before: 20
Thread 0 after: 6955
Thread 1 before: 20
Thread 1 after: 7022
Thread 2 before: 20
Thread 2 after: 3380
```

Vars: примеры

```
(defn run-thread2 [x]
  (.run (fn []
          (binding [*test-var* (rand-int 10000)]
            (println "Thread " x " var=" *test-var*)
            (set! *test-var* (rand-int 10000))
            (println "Thread " x " var2=” *test-var*)))))

user=> (doseq [x (range 3)] (run-thread2 x))
Thread 0 var= 3693
Thread 0 var2= 4408
Thread 1 var= 3438
Thread 1 var2= 2624
Thread 2 var= 6193
Thread 2 var2= 2265
```

Vars: примеры

```
(defn run-thread3 [x]
  (.run (fn []
          (set! *test-var* (rand-int 10000))
          (println "Thread " x " var2=" *test-var*))))
user=> (run-thread3 10)
java.lang.IllegalStateException: Can't change/
establish root binding of: *test-var* with set

user=> *test-var*
20
user=> (alter-var-root #'*test-var* (constantly 10))
10
user=> *test-var*
10
```

Валидаторы

```
user=> (def a (atom 2))
user=> (set-validator! a pos?)
user=> (swap! a dec)
1
user=> (swap! a dec)
IllegalStateException Invalid reference state
clojure.lang.ARef.validate (ARef.java:33)
```

Наблюдатели

```
user=> (def a (atom 1))
#'user/a
user=> (add-watch a "watch 1: "
                  (fn [k r o n] (println k r o n)))
#<Atom@2b36b44e: 1>
user=> (add-watch a "watch 2: "
                  (fn [k r o n] (println k r o n)))
#<Atom@2b36b44e: 1>
user=> (swap! a inc)
watch 1:  #<Atom@2b36b44e: 2> 1 2
watch 2:  #<Atom@2b36b44e: 2> 1 2
2
```

Наблюдатели

```
user=> (remove-watch a "watch 1: ")
#<Atom@372d95a: 1>
user=> (swap! a inc)
watch 2:  #<Atom@372d95a: 2> 1 2
2
user=> (def ^:dynamic b 1)
user=> (add-watch (var b) "dynamic: "
                  (fn [k r o n] (println k r o n)))
user=> (alter-var-root (var b) (constantly 42))
dynamic:  #'user/b 1 42
42
user=> (binding [b 10] (println b))
10
nil
```

Изменяемое состояние (разное)

- Transients (переходные структуры данных)
- Изменяемые поля в `deftype`
- Локальные vars

Transients

```
(defn vrange [n]
  (loop [i 0 v []]
    (if (< i n)
      (recur (inc i) (conj v i))
      v)))

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

user> (time (def v (vrange 1000000)))
"Elapsed time: 189.004 msecs msecs"
user> (time (def v2 (vrange2 1000000)))
"Elapsed time: 99.861 msecs"
```

Изменяемые поля в deftype

```
(defprotocol TestProtocol
  (get-data [this])
  (set-data [this o]))

(deftype Test [^:unsynchronized-mutable x-var]
  TestProtocol
  (set-data [this o] (set! x-var o))
  (get-data [this] x-var))

=> (def a (Test. 10))
=> (get-data a)
10
=> (set-data a 42)
=> (get-data a)
42
```

Локальные vars

- `with-local-vars` позволяет определить локальные vars, с которыми можно работать через `var-set` & `var-get` (или @)

```
(defn factorial [x]
  (with-local-vars [acc 1, cnt x]
    (while (> @cnt 0)
      (var-set acc (* @acc @cnt))
      (var-set cnt (dec @cnt)))
    @acc))
```

Часть 2:

Параллельное и конкурентное программирование

Параллельное выполнение кода

- Встроенные функции:
  - `pmap` – параллельный аналог map
  - pcalls – параллельное вычисление функций
  - pvalues – параллельное вычисление блоков кода

Примеры: pmap, pvalues

```
user=> (defn long-job [n]
         (Thread/sleep 3000)
         (+ n 10))

user=> (time (doall (map long-job (range 4))))
"Elapsed time: 12000.662614 msecs"
(10 11 12 13)
user=> (time (doall (pmap long-job (range 4))))
"Elapsed time: 3001.826403 msecs"
(10 11 12 13)

user=> (time (doall (pvalues
                      (do (Thread/sleep 3000) 1)
                      (do (Thread/sleep 3000) 2)
                      (do (Thread/sleep 3000) 3))))
"Elapsed time: 3000.826403 msecs"
(1 2 3)
```

Futures

- Вычисляются в отдельном потоке
- Результат кешируется
- Доступ через `deref` или `@`
- Блокировка если результата еще нет
- Возможность отмены выполнения

Futures: пример

```
user=> (def future-test
         (future (do (Thread/sleep 10000)
                     :finished)))
user=> @future-test ;; будет ждать результата
:finished
user=> @future-test ;; сразу вернет значение
:finished
```

Delays

- Откладывает выполнение кода до доступа к результату

```
user=> (defn use-delays [x]
         {:result (delay (println "Evaluating
result..." x) x)
          :some-info true})

user=> (def a (use-delays 10))
user=> a
{:result #<Delay@259c3236: :pending>, :some-info
true}
user=> @(:result a) ;; выполняется весь код delay
Evaluating result... 10
10
user=> @(:result a) ;; возвращается только результат
10
user=> (:result a)
<Delay@259c3236: 10>
```

Promises

- Координация между потоками выполнения
- Блокируется при доступе к еще не отправленным данным
- Результат кешируется

```
user=> (def p (promise))
user=> (do (future
             (Thread/sleep 5000)
             (deliver p :fred))
           @p)
```

Блокировки

- `locking` обеспечивает блокировку доступа к объекту

```
=> (defn add-to-map [h k v]
     (locking h
       (.put h k v)))

=> (def h (java.util.HashMap.))
=> (add-to-map h "test" "value")
"value"
=> h
{"test" "value"}
```

Средства JVM: потоки и т.п.

- Легкость вызова кода Java
- Функции без аргументов реализуют интерфейсы `Runnable` & `Callable`

```
=> (.run (Thread.
            #(println "Hello world!")))
Hello world!
nil
```

Часть 3:

Advanced Topics

Reducers

- Введены в Clojure 1.5
- Не создают промежуточных коллекций
- Используют fork/join при выполнении `fold`
- Свои версии функций `map`, `fold`, `filter`, и т.п.
- Ресурсы:
  - http://clojure.com/blog/2012/05/08/reducers-a-library-and-model-for-collection-processing.html
  - http://clojure.com/blog/2012/05/15/anatomy-of-reducer.html
  - http://adambard.com/blog/clojure-reducers-for-mortals/
  - http://www.infoq.com/presentations/Clojure-Reducers

Reducers: пример

```
=> (require '[clojure.core.reducers :as r])
=> (use 'criterium.core)

=> (bench (reduce + (map inc v)))
; Execution time mean : 7.793994 ms
=> (bench (r/reduce + (r/map inc v)))
; Execution time mean : 5.604963 ms
=> (bench (r/fold + (r/map inc v)))
; Execution time mean : 2.095184 ms
```

core.async

- Асинхронное программирование с помощью каналов
- Подобно goroutines в Go
- Поддерживает Clojure & ClojureScript
- Ресурсы:
  - http://clojure.com/blog/2013/06/28/clojure-core-async-channels.html
  - http://stuartsierra.com/2013/12/08/parallel-processing-with-core-async
  - http://swannodette.github.io/2013/07/12/communicating-sequential-processes/
  - http://blog.drewolson.org/blog/2013/07/04/clojure-core-dot-async-and-go-a-code-comparison/
  - http://www.leonardoborges.com/writings/2013/07/06/clojure-core-dot-async-lisp-advantage/
  - http://www.infoq.com/presentations/clojure-core-async
  - http://www.infoq.com/presentations/core-async-clojure

Avout

- Атомы и ссылки в распределенной среде
- Координация через ZooKeeper
- Разные backends для хранения состояния – MongoDB, SimpleDB, плюс возможность расширения
- Можно использовать стандартные функции – deref, наблюдатели, валидаторы
- Собственная версия функций для изменения состояния: `swap!!`, `dosync!!`, `alter!!`, etc.
- Подробно – на http://avout.io/

Avout: пример

```
(use 'avout.core)
(def client (connect "127.0.0.1"))

(def r0 (zk-ref client "/r0" 0))
(def r1 (zk-ref client "/r1" []))

(dosync!! client
  (alter!! r0 inc)
  (alter!! r1 conj @r0))
```

Pulsar

- Реализует различные конкурентные операции
- Включает поддержку акторной модели
- Pattern matching как в Erlang, включая двоичные данные
- Основана на Java библиотеке Quasar
- Ресурсы:
  - http://blog.paralleluniverse.co/2013/05/02/quasar-pulsar/
  - http://puniverse.github.io/pulsar/

Lamina

- Предназначена для анализа потоков данных
- Потоки как каналы
- Возможность параллелизации обработки данных
- Ресурсы
  - https://github.com/ztellman/lamina
  - http://adambard.com/blog/why-clojure-part-2-async-magic/

Hadoop-based

- clojure-hadoop (http://github.com/alexott/clojure-hadoop)
- parkour (https://github.com/damballa/parkour)
- PigPen (https://github.com/Netflix/PigPen)

Ресурсы

- http://java.ociweb.com/mark/stm/article.html
- Clojure Programming by Chas Emerick, Brian Carper, Christophe Grand. O'Reilly, 2012
- http://aphyr.com/posts/306-clojure-from-the-ground-up-state
- http://www.infoq.com/presentations/Value-Identity-State-Rich-Hickey (видео)
- http://skillsmatter.com/podcast/clojure/you-came-for-the-concurrency-right (видео)