



Отложенные вычисления и ПОТОКИ

Денис С. Мигинский

Загадка

```
(letfn [(fact [n]
          (reduce *' (range 1 n)))]
  (time (fact 1000))
  (time (map fact (range 1 1001))))
```

>>

"Elapsed time: 2.779141 msecs"

"Elapsed time: 0.014613 msecs"

;объясните результат профилирования

Разгадка

```
(letfn [(fact [n]
          (reduce * (range 1 n)))]
  (time (fact 1000))
  (time (nth
          (map fact (range 1 1001))
          999)))
```

>>

"Elapsed time: 2.975903 msecs"

"Elapsed time: 588.023119 msecs"

; время затрачено на востребование
; 999-го элемента

Рекурсивное определение последовательности: попытка 1

```
(def naturals  
  (cons 1 (map inc naturals)))
```

```
>>
```

```
#<CompilerException  
java.lang.IllegalStateException: Var  
user/naturals is unbound.
```

Рекурсивное определение последовательности: попытка 2

```
(def naturals  
  (lazy-seq  
    (cons 1 (map inc naturals))))
```

```
(nth naturals 10)
```

```
>>
```

```
11
```

Устройство «ленивых» последовательностей

```
(def naturals  
  (lazy-seq  
    (cons 1 (map inc naturals)))))
```

naturals

→ обещание вычислить `(cons 1 (map ...`

```
(nth naturals 10)
```

```
; вычисляем (cons 1 (map ...
```

naturals

→ **1**

→ обещание `(cons (inc (first naturals)) (map ...`

```
; вычисляем (cons (inc (first naturals)) ...
```

naturals

→ **1**

→ **2**

→ обещание `(cons (inc (second naturals)) (map ...`

```
; ...
```

Бесконечная последовательность (поток) чисел Фибоначчи

;рекурсивное определение

```
(def fibs
  (lazy-cat '(0 1)
    (map + fibs (rest fibs))))
```

;определение через порождающую функцию

```
(let [fibs (map
              first
              (iterate
                (fn [[v1 v2]]
                  [v2 (+ v1 v2)])
                [0 1]))]
  (nth fibs 10))
```

«Ленивые» операции над последовательностями

; порождение «ленивой»

; последовательности из коллекции
(seq coll)

; порождение «обещания» вычислить

; последовательность
(lazy-seq & body)

; эквивалентно (concat (lazy-seq s1)

; (lazy-seq s2) ...

(lazy-cat s1 s2 & rest)

«Ленивые» операции над последовательностями

;порождение (x (f x) (f (f x)) ...
(iterate f x)

;получение части последовательности
(take n coll)

(for ...
(map ...
(filter ...
(rest ...

«Неленивые» операции

;выбор элемента последовательности

(nth coll n)

(first coll)

...

;принудительное вычисление

(doall coll)

(dorun coll)

;аналогично for, но допускает побочные

;эффекты и не возвращает значения.

;Похоже на each в Ruby.

(doseq seq-expr & body)

(reduce ...

(cons ...

Потоки

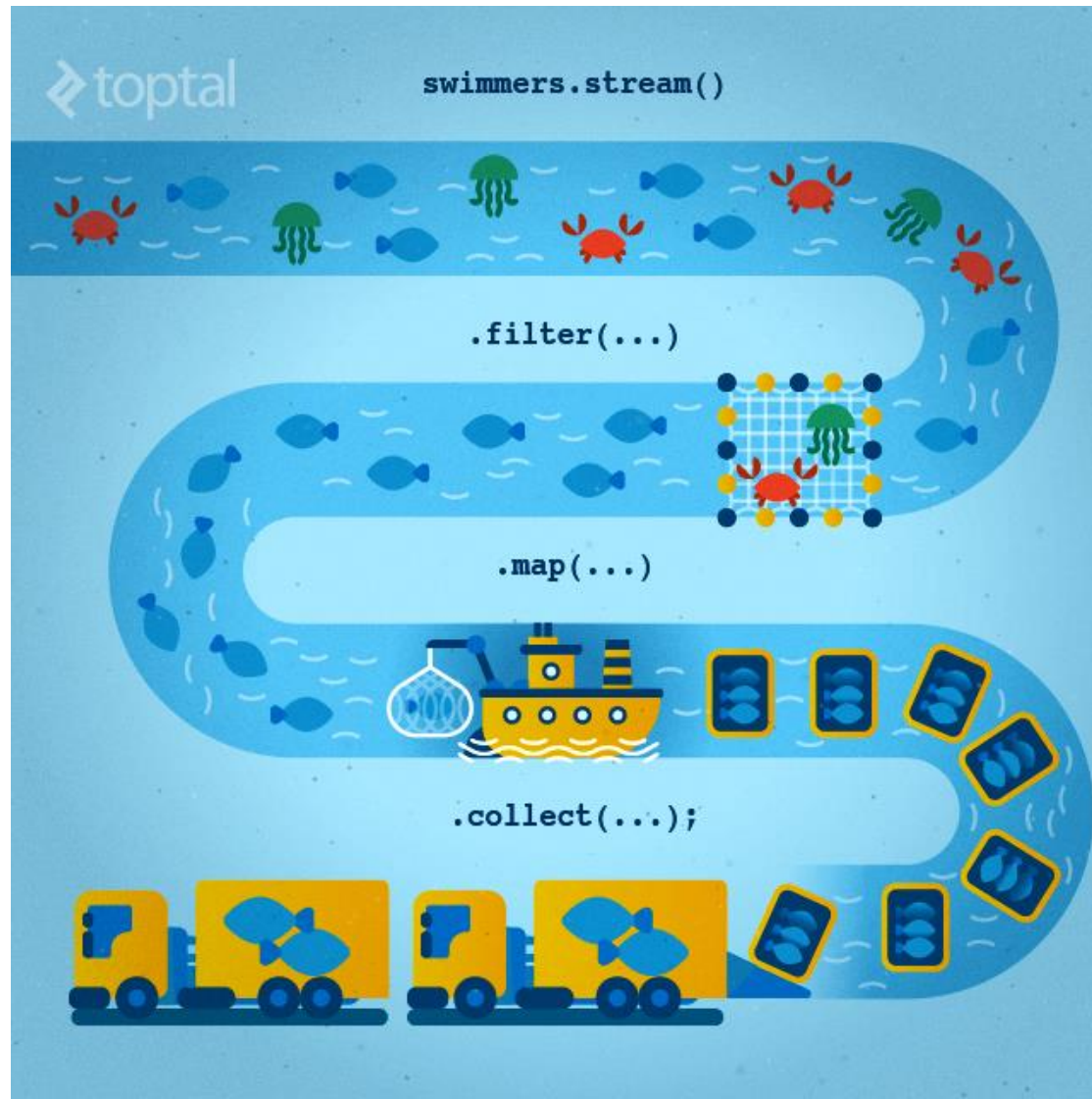
```
;; сколько элементов вычислится  
;; в map и filter?
```

```
(take 10  
  (map #(* % %)  
        (filter #(= 0 (mod % 2))  
                (range 100))))
```

```
;; эквивалентная запись
```

```
(->> (range 100)  
      (filter #(= 0 (mod % 2)))  
      (map #(* % %))  
      (take 10))
```

Иллюстрация (Java)



Файловые потоки

```
(with-open [fl (clojure.java.io/reader  
                "/tmp/my_file.txt")]  
  (->> (line-seq fl)  
        (map #(Integer/parseInt %))  
        (filter #(= 0 (mod % 2)))  
        (map #(* % %))  
        (take 10)))
```

Представление времени и состояния

	Императивное программирование	Функциональное программирование
Модельное состояние	Состояние переменных	Неизменяемая структура данных
Изменение модельного состояния	Изменение состояния переменных	Порождение новой структуры данных в потоке (stream)
Модельное время время	Тожественно вычислительному времени	Перемещение по потоку

Функциональные объекты с операцией отката

Задача: необходимо реализовать универсальное (т.е. не привязанное к конкретной задаче/предметной области) представление объектов с операцией отката (undo)

Анализ задачи

Механизм не должен раскрывать детали своей реализации (принцип абстракции)

Механизм не должен ничего знать про структуру состояния объекта и набор операций над ним (требование универсальности)

Необходимо предоставлять прозрачный доступ к объекту:

- инициализация объекта
- получение состояния объекта
- изменение состояния объекта

Должна быть реализована операция undo

Проектирование: API

;порождение объекта

```
(defn object [init-state] ...)
```

;получение состояния объекта

```
(defn state [obj] ...)
```

;замена состояния объекта

```
(defn replace-state [obj new-state]...)
```

;изменение состояния мутатором

```
(defn change-state [obj mutator] ...)
```

;возврат в предыдущее состояние

```
(defn undo [obj] ...)
```

Реализация

```
(defn object [state]  
  (list state))
```

```
(defn state [obj]  
  (first obj))
```

```
(defn replace-state [obj new-state]  
  (cons new-state obj))
```

```
(defn change-state [obj mutator]  
  (replace-state obj (mutator (state obj))))
```

```
(defn undo [obj]  
  (rest obj))
```

Поккрытие тестами

```
(test/is (= (state (object 1)) 1))
(test/is (= (state
             (replace-state (object 1) 2))
           2))
(test/is (= (state
             (change-state (object 1) inc))
           2))
(test/is (= (state
             (undo
              (replace-state (object 1) 2)))
           1))
; особые случаи
(test/is (= (state (undo (object 1))) nil))
(test/is (= (state
             (replace-state
              (undo (object 1)) 2))
           2))
```

Вариации на тему отложенных вычислений

Крайние случаи:

- вычисление происходит в момент подачи команды (чисто императивный случай)
- вычисление происходит в момент востребования значения (чисто функциональный случай)


Промежуточный (асинхронный) случай:

- в момент подачи команды запускается отдельный поток исполнения (возможно, на удаленном узле)
- в момент востребования результата выбирается результат вычисления (при необходимости с ожиданием)
- могут быть предоставлены дополнительные средства для мониторинга вычислений



Задача 2.2 поток простых чисел

Реализуйте бесконечную последовательность простых чисел



Задача 3.2: численное интегрирование с потоками

Модифицировать решение задачи 3.1 таким образом, чтобы вместо мемоизации использовались потоки

Показать прирост производительности с помощью **time**.
Обеспечить покрытие функциональными тестами.