Лабораторная работа #4

Дисциплина: "Функциональное программирование"

Дата: 2021/12

Выполнили: Федоров Сергей | Лазурин Евгений, P34113

Название: "Nats Streaming, Clojure eDSL"

Цель работы: Получить навыки работы со специфичными для выбранной технологии/языка программирования приёмами.

Вариант:

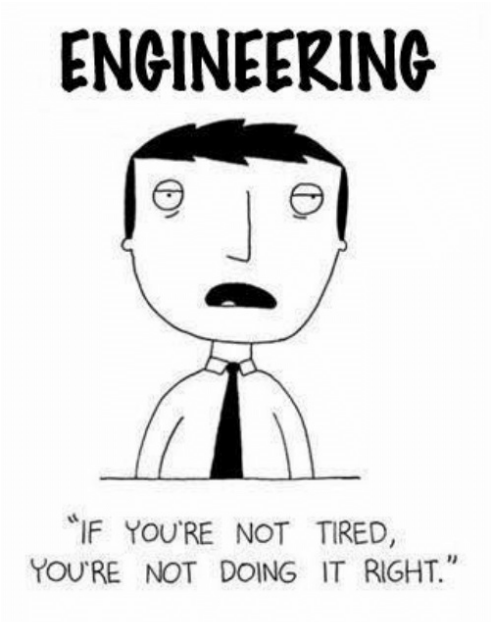
ЯП	Clojure
Подход	eDSL
Область применения	Клиент nats.io

Требования:

1. Должны быть реализованы клиенты для producer'a и subscriber'a
2. Взаимодействие конфигурация и взаимодействие с библиотекой построено на базе eDSL
3. Поддерживается стриминг данных

Выполнение

Ссылка на репозиторий



О программе

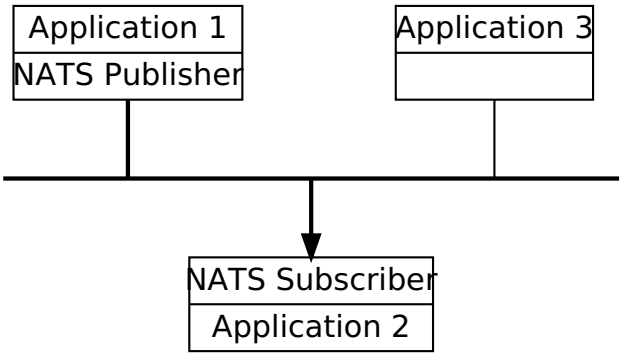
О целевой системе

Пишется клиент для nats.io, соответственно рассмотрим концепцию работы с данный системой:

Система сообщений nats, представляет собой "простую" общую шину, по которой можно передавать данные с разделением по префиксным строчкам. При настроенном кластере, не имеет значение конфигурация сети, а это означает что с точки зрения application layer, нужно знать лишь соответствующий топик и адрес одного из bootstrap-серверов Nats.

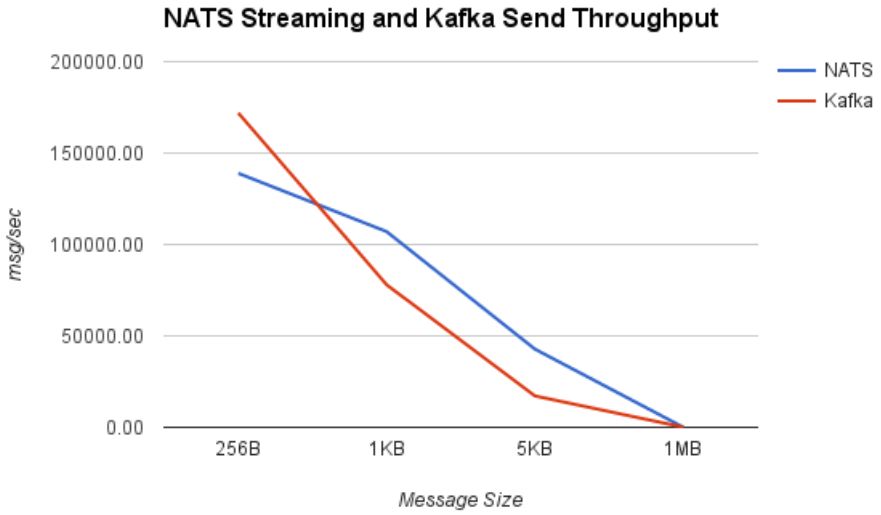
Сообщения, это закодированная информация в виде бинарного массива, объединённая вместе с соответствующей мета датой.

Схема сообщений:



Сообщения поступают в систему от Publisher'ов, и имеющиеся в данный момент Subscriber'ы, по префиксу получают эти сообщения.

Такая свободная архитектура имеет свои последствия. Например, там где kafka может оптимизировать использование некоторых ресурсов, nats-streaming такой возможности не имеет. Это играет соответствующую роль в плане производительности:



Однако как мы видимо, разница не настолько драматичная, как нам могло бы показаться...

Логика работы программы (и с программой)

Взаимодействие с библиотекой происходит посредством **eDSL** (**e**mbedded **D**omain-**S**pecific **L**anguage):

1. Каждый клиент должен создать NatsConnection:

```
(connection
 (with :verbose)
 (with :server address)
 ; Many more other parameters]
 (with :connection-name "clojure-test"))
```

2. Для публикации нового сообщения мы можем использовать метод `publish`:

```
(publish
    (to connection)
    (subject "test-topic")
    (message "Hello world!")
    (headers {"authors" ["Sergey" "Eugene"]})
    sync-pub)
```

для публикации можно использовать 4 разных режима: * `sync-pub` - синхронно * `async-pub` - асинхронно на thread-pool'е * `future-pub` - асинхронно, обернув в `future` * `stream-pub` - создать стрим типа `sink`, и добавляя туда сообщения коммитить их в nats

3. Для получения новых сообщений мы можем использовать метод `subscribe`:

```
(subscribe
    (to connection_sub)
    (subject topic)
    sync-sub)
```

при подписке можно также использовать 4 разных режима: * `sync-sub` *в ручную* - возвращает `nats.Subscription` и мы используем `.nextMessage` * `sync-sub + stream-sub` - возвращает синхронную `clojure.lazy-seq` коллекцию * `async-sub + fn callback` - создает подписку с уже протянутым внутрь callback'ом * `async-sub + stream-sub` - возвращает асинхронный стрим сообщений (закрывает подписку на `:drained`)

Программная реализация

Структура проекта:

```
.
├── project.clj
├── docker-compose.yaml
└── src
      └── functional_programming_itmo_2021
          └── lab_4
               ├── main.clj
               ├── nats
               │   ├── connection.clj
               │   ├── message.clj
               │   └── operations.clj
               └── util.clj
```

Настройка Clojure-окружения с помощью Leiningen:

`project.clj`

```
(defproject functional-programming-itmo-2021 "1.0.0"
            :dependencies [[org.clojure/clojure "1.10.3"]
                           [nrepl/lein-nrepl "0.3.2"]
                           [org.clojure/core.async "1.5.640"]
                           [io.nats/jnats "2.13.1"]
                           [org.clojure/core.match "1.0.0"]
                           [manifold "0.2.3"]]
            :profiles {
                       :lab_4 {
                               :repl-options {
                                              :init-ns functional-programming-itmo-2021.lab-4.main
                                              :package functional-programming-itmo-2021.lab-4
                                              }
                               :main functional-programming-itmo-2021.lab-4.main
                               }
                       }
            )
```

Старт локального nats-server:

`docker-compose.yaml`

```
version: '3'

services:
  local_nats:
    image: nats:latest
    ports:
      - 4222:4222
      - 8222:8222
```

Вспомогательные функции:

`util.clj`

```
(ns functional-programming-itmo-2021.lab-4.util
  (:import (java.time Duration)))

(defmacro flat-map [f & arguments]
  `(flatten (map ~f ~@arguments)))

(defprotocol DurationLike
  (duration ^Duration [this]))

(extend-protocol DurationLike
  Duration
  (duration [this] this)

  Long
  (duration [this] (Duration/ofSeconds this)))
```

Протокол сообщений (весьма примитивный):

`message.clj`

```
(ns functional-programming-itmo-2021.lab-4.nats.message)

(defprotocol NatsMessageLike
  (to-dto [this]))

(extend-protocol NatsMessageLike
  String
  (to-dto [str] (.getBytes str "UTF-8")))
```

Описание операции установки соединения:

`connection.clj`

```
(ns functional-programming-itmo-2021.lab-4.nats.connection
  (:require [functional-programming-itmo-2021.lab-4.util :as u]
            [clojure.core.match :refer [match]])
  (:import (io.nats.client Options$Builder ConnectionListener ReconnectDelayHandler Nats JetStreamOptions$Builder)
           (javax.net.ssl SSLContext)))


(defn with [^Options$Builder builder & arguments]
  (match (vec arguments)
    [:no-echo]          (.noEcho builder)
    [:no-headers]       (.noHeaders builder)
    [:no-no-responders] (.noNoResponders builder)
    [:no-reconnect]     (.noReconnect builder)

    [:auth-handler creds-file] (.authHandler builder (Nats/credentials creds-file))
    [:auth-handler jwt nkey]   (.authHandler builder (Nats/credentials jwt nkey))

    [:server server]            (.server builder server)
    [:servers & servers]        (.servers builder (into-array String servers))
    [:ping-interval dur]        (.pingInterval builder (u/duration dur))
    [:ssl-context ssl-type]     (.sslContext builder (SSLContext/getInstance ssl-type))
    [:token token]              (.token builder token)
    [:data-port-type port-type] (.dataPortType builder port-type)

    [:connection-name name]     (.connectionName builder name)
    [:connection-listener func] (.connectionListener builder (reify ConnectionListener
                                                               (connectionEvent [_ conn type] (func conn type))))
    [:connection-timeout dur]   (.connectionTimeout builder (u/duration dur))

    [:reconnect-buffer-size size]  (.reconnectBufferSize builder size)
```

```
    [:reconnect-buffer-size size]   (.reconnectBufferSize builder size)
    [:reconnect-delay-handler func] (.reconnectDelayHandler builder (reify ReconnectDelayHandler
                                                                      (getWaitTime [_ totalTries] (func totalTries))))
    [:reconnect-jitter dur]         (.reconnectJitter builder (u/duration dur))
    [:reconnect-jitter-tls dur]     (.reconnectJitterTls builder (u/duration dur))
    [:reconnect-wait dur]           (.reconnectWait builder (u/duration dur))

    [:max-control-line limit]               (.maxControlLine builder limit)
    [:max-messages-in-outgoing-queue limit] (.maxMessagesInOutgoingQueue builder limit)
    [:max-pings-out limit]                  (.maxPingsOut builder limit)
    [:max-reconnects limit]                 (.maxReconnects builder limit)

    [:request-cleanup-interval dur] (.requestCleanupInterval builder (u/duration dur))
    [:buffer-size size]             (.bufferSize builder size)
    [:user-info user-name pass]     (.userInfo builder user-name pass)
    [:inbox-prefix prefix]          (.inboxPrefix builder prefix)

    [:verbose]                        (.verbose builder)
    [:discard-message-when-full]      (.discardMessagesWhenOutgoingQueueFull builder)
    [:old-request-style]              (.oldRequestStyle builder)
    [:pedantic]                       (.pedantic builder)
    [:open-tls]                       (.opentls builder)
    [:secure]                         (.secure builder)
    [:trace-connection]               (.traceConnection builder)

    [:executor service] (.executor builder service)
    [:error-listener error-listener] (.errorListener builder error-listener)
    ))

(defmacro connection
  "Creates a connection to NATS mesh-server.
  Configuration is achieved through eDSL grammar:

      (connection
        (with :no-headers)
        (with :no-echo)
        (with :buffer-size 100)
        (with :max-reconnects 5)
        ...
        (with :verbose))

  Available keyword properties:
    :no-echo :no-headers :no-no-responders :no-reconnect
    :auth-handler
    :server :servers :ping-interval :ssl-context :token :data-port-type
    :connection-name :connection-listener :connection-timeout
    :reconnect-buffer-size :reconnect-delay-handler :reconnect-jitter :reconnect-jitter-tls :reconnect-wait
    :max-control-line :max-messages-in-outgoing-queue :max-pings-out :max-reconnects
    :request-cleanup-interval :buffer-size :user-info :inbox-prefix
    :verbose :discard-message-when-full :old-request-style :pedantic n:open-tls :secure :trace-connection
    :executor :error-listener
  "
  [& params]
  `(-> (Options$Builder.) ~@params .build Nats/connect))
```

Описание операций взаимодействия с шиной:

`operations.clj`

```
(ns functional-programming-itmo-2021.lab-4.nats.operations
  (:require [functional-programming-itmo-2021.lab-4.nats.message :as m]
            [clojure.core.async :as a]
            [manifold.stream :as s]
            [functional-programming-itmo-2021.lab-4.util :as u])
  (:import (io.nats.client Connection Subscription Dispatcher MessageHandler Message)
           (io.nats.client.impl NatsMessage Headers)
```

```
           (io.nats.client.impl NatsMessage Headers)
           (clojure.lang Fn IPersistentMap)
           (java.time Duration)))

(defmacro ^:private edit-aggregate
  ([keyword key-name] `(edit-aggregate ~keyword ~key-name identity))
  ([keyword key-name opt-f]
   `(fn [aggregate# elem#]
     (if-not (nil? (~keyword aggregate#))
       (-> (str "Already defined " ~key-name) IllegalArgumentException. throw)
       (assoc aggregate# ~keyword (~opt-f elem#))))))

(defn- check-aggregate [aggregate]
  (let [assert-not-nil #(-> aggregate %2 nil? not (assert (str %1 " is nil")))]
    (assert-not-nil "connection" :conn)
    (assert-not-nil "subject" :subject)))


(defrecord PublishAggregate [^Connection conn
                                       ^String subject
                                       ^ String reply-to
                                       #^bytes message
                                       ^IPersistentMap headers])
(def empty-pub (->PublishAggregate nil nil nil nil nil))

(defn- produce-msg [^PublishAggregate aggregate]
  (let [subject (:subject aggregate)
        reply (:reply-to aggregate)
        data (:message aggregate)
        headers (:headers aggregate)]
    (NatsMessage.
      subject
      reply
      (if (nil? headers) nil (reduce-kv #(.add %1 %2 %3) (Headers.) headers))
      data
      true
      ))
  )

; Publish DSL

(def to (edit-aggregate :conn "connection"))
(def subject (edit-aggregate :subject "subject" #(str %)))
(def reply-to (edit-aggregate :reply-to "reply-to" #(str %)))
(def message (edit-aggregate :message "message" #(m/to-dto %)))
(def headers (edit-aggregate :headers "headers"))

(defn sync-pub [^PublishAggregate aggregate]
  (check-aggregate aggregate)
  (.publish (:conn aggregate) (produce-msg aggregate)))

(defn async-pub [^PublishAggregate aggregate] (a/go (sync-pub aggregate)))

(defn future-pub [^PublishAggregate aggregate]
  (check-aggregate aggregate)
  (let [comp-future (.request (:conn aggregate) (produce-msg aggregate))]
    (future (.get comp-future))))

; Subscribe DSL

(defrecord SubscribeAggregate [^Connection conn ^String subject])
(def empty-sub (->SubscribeAggregate nil nil))

(defn ^Subscription sync-sub [^SubscribeAggregate aggregate]
  (check-aggregate aggregate)
  (.subscribe (:conn aggregate) (:subject aggregate)))
```

```
(defn async-sub
  ([^SubscribeAggregate aggregate]
   (check-aggregate aggregate)
   (.createDispatcher (:conn aggregate)))
  (^Subscription [^SubscribeAggregate aggregate ^Fn fn]
   (.subscribe (async-sub aggregate) (proxy [MessageHandler] []
                                       (onMessage [^Message msg] (fn msg))))))

(defn stream-sub
  ([^Dispatcher dispatcher subject]
   (let [stream (s/stream)
         source (s/source-only stream)
         handler (proxy [MessageHandler] []
                   (onMessage [msg] (s/put! stream msg)))
         subscription (.subscribe dispatcher subject handler)]
     (s/on-closed stream #(.close subscription))
     source))
  ([^Subscription subscription]
   (lazy-seq (cons
              (.nextMessage subscription (u/duration 10))
              (stream-sub subscription)))))

(defn stream-pub
  "returns a Manifold sink-only stream which publishes items put on the stream
   to NATS"
  ([^PublishAggregate aggregate]
   (check-aggregate aggregate)
   (let [stream (s/stream)]
     (s/consume (fn [msg hval]
                  (let [adjusted-agg (-> aggregate
                                         (message msg)
                                         (headers hval))]
                    (.publish (:conn aggregate) msg)))
                stream)
     (s/sink-only stream))))

(defmacro publish
  "Publish a message to NATS.
  Configuration is derived from eDSL grammar:

      (publish
        (to nats-connection)
        (subject \"some fonny subject\")
        (reply-to \"somewhere to put answer to\")
        (message \"where important thing to share\")
        (headers {\"regions\" [\"Central Asia\" \"Central Europe\"]}
        ; Position dependent statements
        (sync-pub)
        ; or (async-pub)
        ; or (future-pub)
        ; or (stream-pub) message and headers are defined on consumer
  "
  [& body]
  `(-> empty-pub ~@body))

(defmacro subscribe
  "Subscribe to a subject at NATS.
  Configuration is derived from eDSL grammar:

      (subscribe
        (to nats-connection)
        (subject \"some fonny subject\"))
        (sync-sub)
        [(stream-sub)]
```

```
        ; or (async-sub [callback func])
             [(stream-sub)]

  "
  [& body]
  `(-> empty-sub ~@body))
```

Входная точка и демонстрация работы:

`main.clj`

```
(ns functional-programming-itmo-2021.lab-4.main
   (:require [functional-programming-itmo-2021.lab-4.nats.operations :refer :all]
      [functional-programming-itmo-2021.lab-4.nats.connection :refer :all]
      [manifold.stream :as s])
   (:import (io.nats.client Message)))


(def topic "some-topic")
(def address "localhost:4222")

(def connection_pub
   (connection
      (with :verbose)
      (with :server address)
      (with :connection-name "clojure-test-publish")))

(def connection_sub
   (connection
      (with :verbose)
      (with :server address)
      (with :connection-name "clojure-test-subscribe")))

(def subscription
   (subscribe
      (to connection_sub)
      (subject topic)
      async-sub
      (stream-sub topic)
      ))

(defn do-pub []
   (repeatedly 10 #(publish
                     (to connection_pub)
                     (subject topic)
                     (message "Hello world!")
                     (headers {"authors" ["Sergey" "Eugene"]})
                     sync-pub)))

(defn -main []
   (println "Started")
   (doall (do-pub))
   (println "Published 10 messages")
   @(s/consume (fn [^Message msg]
                 (let [data (.getData msg)
                       text (String. data "UTF-8")]
                    (println "Received a message: " text)))
               subscription))
```

`stdout`

```
 Started
Published 10 messages
Received a message:  Hello world!
Received a message:  Hello world!
Received a message:  Hello world!
Received a message:  Hello world!
Received a message:  Hello world!
Received a message:  Hello world!
Received a message:  Hello world!
Received a message:  Hello world!
Received a message:  Hello world!
Received a message:  Hello world!
```

Что получилось по итогу (a.k.a. выводы)

Что есть: 1. Была разработана система взаимодействия с nats.io 2. Поддерживается множество различных способ исполнения для публикации или получения сообщений 3. Поддерживаются все параметры конфигурации соединения, в идиоматическом стиле 4. Все взаимодействие с системой производится с помощью eDSL, а за ее рамками с помощью `clojure.core` или `clj-commons` 5. eDSL был написан с совместным использованием `defn` и `defmacro`, что является идиоматичным стилем. *(тут кстати легко споткнуться) 6. Предположительно данное решение, даже лучше существующих OSS аналогов на `clojure`

Мысли:

Были интересно, но достаточно непросто въехать в механизм построения DSL в clojure. Подход своеобразный и в какой-то момент, когда все сделано правильно, выглядит как очень приятная магия. Однако до сих пор присутствует уверенность, что использование статических языков по типу Scala или Haskell с авто-определением типов и type-checking'ом, позволяет и писать и использовать DSL в более приятном виде, особенно в процессе разработки.

(А вообще, лучше словами. Тут много эмоций...)