F [SCALA]

АНДРЕЙ
ЛИСТОПАДОВ

CLOJURE.CORE.ASYNC - DEEP DIVE

Обо мне

- Clojure инженер в Health Samurai
- Работал над проектами со смесью Scala и Clojure
- Люблю асинхронщину
- Участвую в разработке языка Fennel

HEALTH SAMURAI

Telegram: **@aorst**
Git[La|Hu]b: **@andreyorst**

О чем сегодня поговорим

- Асинхронное программирование
  - Зачем/Почему
  - История появления асинка в Clojure
- Устройство clojure.core.async
  - Общая модель организации кода
  - Устройство асинхронных тредов
  - Устройство каналов
  - Таймеры
  - Планировщик/Event Loop
- Портируемость
- Проблемы

Clojure

- Clojure это:
  - функциональный
  - динамический
  - строго-типизированный
  - лисп
- Hosted язык
  - Официально поддерживает JVM, JavaScript, .NET
  - Неофициально реализован над Erlang, Dart, LLVM

Синтаксис Clojure

```clojure
(defn my-function [arg-1, arg-2]
  (let [sum (+ arg-1 arg-2)]
    (println "result:" sum)
    {"result" sum, "args" [arg-1 arg-2]}))


(my-function 40, 2)
;; => {"result" 42, "args" [40 2]}
```

Clojure

```javascript
function my_function (arg1, arg2) {
  let sum = arg1 + arg2
  console.log("result: ", sum)
  return { "result": sum, "args": [arg1, arg2] }
}

my_function(40, 2)
// => {"result": 42, "args": [40, 2]}
```

JavaScript

Асинхронное программирование

- Async в языке либо
  - Есть сразу
  - Приделан “сбоку”
- Async в JVM
  - Green Threads (до JDK версии 1.1)
  - ...ничего...
  - Virtual Threads, начиная с JDK21
- Async на других платформах
  - BEAM - actor model
  - Go - CSP
  - .NET - async/await
  - Lua - coroutines

Parallelism VS Async – в чем разница?

- Параллелизм
  - Подходит для CPU-bound задач
  - Одновременное выполнение
  - Ограничено количеством вычислительных ядер
- Асинхронность
  - Подходит для IO-bound задач
  - Кооперативное выполнение
    - Ограничено допустимой скоростью реакции

Async в Clojure

- Агенты
  - Агент – это “место” для хранения и модификации данных
  - Состояние агента модифицируются отправкой сообщения с “заданием”
  - Задания - чистые функции
  - Сообщения сериализуются, данные изменяются конкурентно-безопасным способом
- Сторонние решения
  - Manifold – event-driven
  - Promesa – async/await
  - Другие...

Проблемы с агентами

- Задания должны быть достаточно короткими
- Блокировка агентов может:
  - Израсходовать тредпул - новые задания не будут выполняться
  - В конечном счете привести к OOME - нет backpressure
- Два отдельных тредпула:
  - Compute: неограниченный для блокирующих заданий
  - IO: Ограниченный, для неблокирующих заданий
- Обработка ошибок
  - В сложных сетях коммуникаций агентов сложно понять, где действительно произошла ошибка
  - Менеджмент состояния агентов добавляет сложность в код
- Не переносится на все поддерживаемые рантаймы

clojure.core.async

- Универсальная модель для асинхронного программирования в Clojure и диалектах
- Основана на CSP (Communicating Sequential Processes)
- Собственные асинхронные треды
- Каналы, как точка синхронизации и средство общения процессов

Каналы

- Небуферизированные:
  - Поддерживают backpressure
- Буферизированные каналы
- Разные стратегии буферизации:
  - Фиксированный: backpressure, если буфер переполнен
  - Оконный сдвиг: всегда хранит **N** самых свежих сообщений
  - Отбрасывание: всегда хранит **N** самых старых сообщений
  - Своя стратегия: выкидывает случайные, например

С каналом вы можете!

- Положить данные! >!
- Забрать данные! <!
- Положить данные, блокируя!! >!!
- Забрать данные, блокируя!! <!!

Блокирующие операции

- Блокируют тред, в котором выполняются
- Не рекомендуются к использованию в асинхронных тредах

```
1.  (def ch (chan 10))
2.  (>!! ch "F[Scala]") ;; положит "F[Scala]" в буфер канала
3.  (println (<!! ch)) ;; Выведет F[Scala] в stdout
```


Неблокирующие операции

- Не блокируют тред
- Можно пользоваться исключительно в асинхронных тредах
  - Является ошибкой компиляции, при использовании вне асинхронных тредов

```
1.  (def ch (chan))
2.  (go
3.    (println (<! ch)))
4.  (go
5.    (>! ch "F[Scala]"))
```


Асинхронные треды

- Собственная реализация виртуальных тредов
- Не зависят от платформы
- Очень лёгкие
- Быстро создаются

Макросы

- Компиляция кода в Clojure выполняется в несколько этапов:
  - Чтение
  - Раскрытие макросов
  - Компиляция
- Макросы, как микро-расширения компилятора
- Имеют все те же возможности, что и обычный код в рантайме
- Принимают на вход **код**, возвращают **код**
  - Код для макроса – обычные **данные**: списки, массивы, таблицы, строки и т.д.

go треды

- Асинхронные треды – вовсе не треды
- **go** – это макрос, трансформирующий код

```
1.  (def ch1 (chan))
2.  (def ch2 (chan))
3.  (go
4.    (let [data (<! ch1)]
5.      (>! ch2 (foo data))))
```


go макрос

```
1.    (let [c (chan 1)
2.          captured-bindings (getThreadBindingFrame)
3.          f (fn state-machine
4.              ([] (aset-all! (AtomicReferenceArray. (int 6)) 0 state-machine 1 1))
5.              ([state]
6.               (let [old-frame (getThreadBindingFrame)
7.                     ret (try
8.                           (resetThreadBindingFrame (aget-object state 3))
9.                           (loop []
10.                            (let [result
11.                                  (case (int (aget-object state 1))
12.                                    1 (take! state 2 ch1)
13.                                    2 (let [data (foo (aget-object state 2))]
14.                                        (put! state 3 ch2 data))
15.                                    3 (return-chan state (aget-object state 2)))]
16.                              (if (identical? result :recur) (recur) result)))
17.                          (catch Throwable ex
18.                            (aset-all! state 2 ex)
19.                            (if (seq (aget-object state 4))
20.                              (aset-all! state 1 (first (aget-object state 4)))
21.                              (throw ex))
22.                            :recur)
23.                          (finally
24.                            (aset-object state 3 (getThreadBindingFrame)) (resetThreadBindingFrame old-frame)))]
25.                 (if (identical? ret :recur)
26.                   (recur state)
27.                   ret))))
28.          state (aset-all! (f) USER-START-IDX c BINDINGS-IDX captured-bindings)]
29.      (run-state-machine-wrapped state)
30.      c)
```

```
(def ch1 (chan))
(def ch2 (chan))
(go
  (let [data (<! ch1)]
    (>! ch2 (foo data))))
```


go макрос

```
1.  (let [c (chan 1) ;; ❶ создаем канал go-блока
2.        captured-bindings (getThreadBindingFrame) ;; ❷ сохраняем текущие thread-local'ы
3.        f (fn state-machine ;; создаем стейт машину
4.            ([] (aset-all! (AtomicReferenceArray. (int 6)) 0 state-machine 1 1)) ;; ❸ инициализация стейт машины, возвращаем стейт
5.            ([state]
6.             (let [old-frame (getThreadBindingFrame) ;; ❹ текущие thread-local'ы на момент старта стейт машины
7.                   ret (try
8.                         (resetThreadBindingFrame (aget-object state 3)) ;; устанавливаем thread-local'ы сохраненные в ❷
9.                         (loop [] ;; ❺ ядро стейт машины
10.                          (let [result
11.                                (case (int (aget-object state 1)) ;; ❻ получение текущего стейта и прыжок
12.                                  1 (take! state 2 ch1)
13.                                  2 (let [data (foo (aget-object state 2))]
14.                                      (put! state 3 ch2 data))
15.                                  3 (return-chan state (aget-object state 2)))]
16.                            (if (identical? result :recur) (recur) result))) ;; ❼ нужно ли продолжать выполнение прямо сейчас
17.                         (catch Throwable ex ;; ❽ обработка ошибок
18.                           (aset-all! state 2 ex)
19.                           (if (seq (aget-object state 4))
20.                             (aset-all! state 1 (first (aget-object state 4)))
21.                             (throw ex))
22.                           :recur)
23.                         (finally ;; ❾ восстановление thread-local'ов сохраненных в ❶
24.                           (aset-object state 3 (getThreadBindingFrame)) (resetThreadBindingFrame old-frame)))]
25.               (if (identical? ret :recur)
26.                 (recur state)
27.                 ret))))
28.        state (aset-all! (f) USER-START-IDX c BINDINGS-IDX captured-bindings)] ;; ❿ сохранение стейта из шага ❸
29.    (run-state-machine-wrapped state)
30.    c)
```

```
(def ch1 (chan))
(def ch2 (chan))
(go
  (let [data (<! ch1)]
    (>! ch2 (foo data))))
```


go макрос

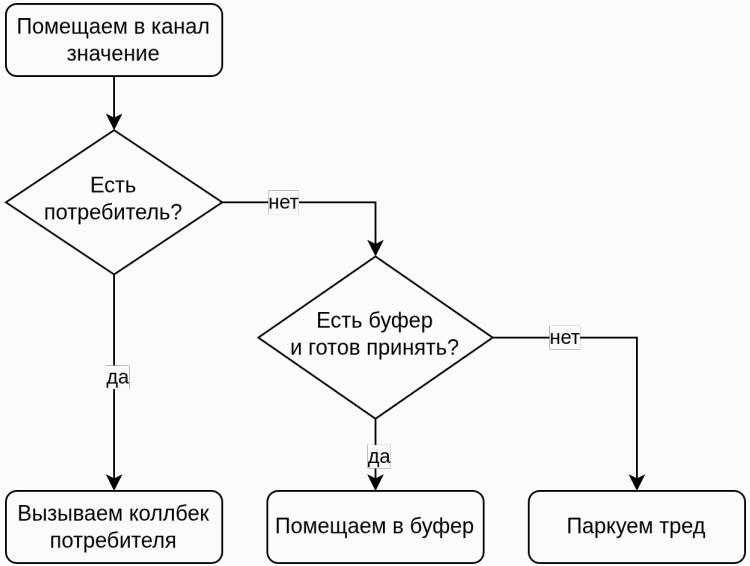
- Компилируется в стейт машину
  - Стратегия Inversion Of Control (IOC)
  - По своей сути напоминает классический **generator** с **yield**
- Операции над каналами преобразуются в **callback**'и
- Запускается специальным планировщиком

```
1.  (defn put! [state blk c val]
2.    (if-let [cb (impl/put! c val
3.                           (fn-handler
4.                              ;; анонимная функция, выполнится если кто-нибудь вызовет take! на канале c
5.                            (fn [ret-val]
6.                              ;; изменяет состояние стейт машины
7.                              (aset-all! state 2 ret-val 1 blk)
8.                              ;; запускает стейт машину
9.                              (run-state-machine-wrapped state))))]
10.       (do (aset-all! state 2 @cb 1 blk)
11.           :recur)
12.       nil))
```

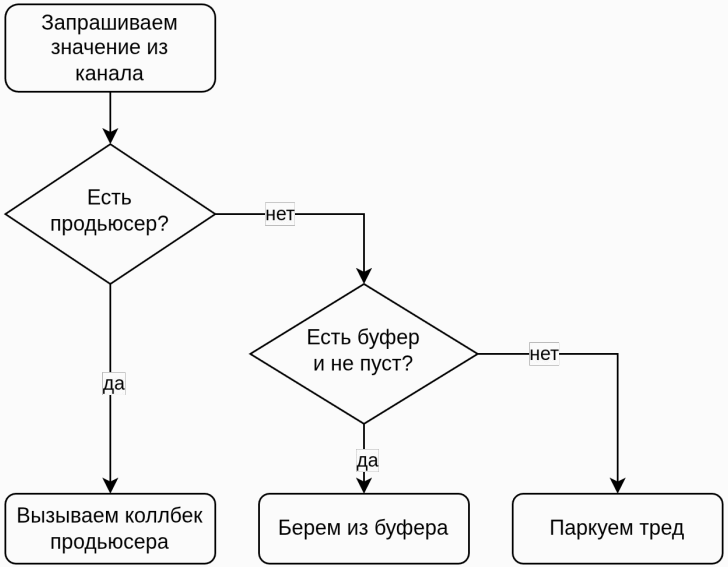

Каналы

- Объект с несколькими очередями
  - Очередь для **put** задач
  - Очередь для **take** задач
  - Буфер для **значений**

Каналы - поведение



помещение в канал



взятие из канала

Таймеры

- Timeout
  - Специальный вид канала
  - Автоматически закрывается
- Мягкие и разделяемые
  - Таймеры, созданные в пределах 10ms используют общий объект
  - Нельзя закрывать таймер вручную, т.к. это может быть чужой таймер

```
1.  (def timer
2.    (timeout 1000)) ;; закроется через 1 секунду
```


Пример использования: ожидание с таймаутом

```
1.  (def ch (chan))
2.
3.  (go
4.    (let [t (timeout 1000)
5.          [data result-chan] (alts! [ch t])] ;; выполняем select на двух каналах, получаем
6.                                             ;; результат и канал из которого его получили
7.
8.      (cond (= result-chan t)   ;; если результирующий канал равен каналу с таймером,
9.            (println "timeout") ;; то произошел таймаут
10.
11.           (= data nil) ;; иначе, если пришел NULL, то кто-то закрыл канал ch
12.           (println ch "was closed")
13.
14.           :else ;; в остальных случаях данные успешно получены
15.           (println data))))
```


Нужен ли для этого всего планировщик?

- Задачи планировщика:
  - Мониторить таймеры
  - Подталкивать асинхронные треды, у которых нет внешних потребителей
- При использовании тредпула, планировщик может:
  - Сократить время отклика в IO-задачах
  - (В меру) смешивать блокирующий и асинхронный код
- Однако, планировщик не обязателен для:
  - Полностью реактивного контекста использования
  - При отказе от использования тредов без внешних потребителей
    - Но только в однопоточных системах

Примеры с планировщиком и без

```
1.  (go
2.    (<! (timeout 1000))
3.    (println "done"))
```

Нет внешнего потребителя
(нужен планировщик, иначе
никогда не выполнится)

```
1.  (def ch (chan))
2.
3.  (go
4.    (<! (timeout 1000))
5.    (>! ch "done"))
6.
7.  (println (<!! ch))
```

Есть внешний потребитель
(планировщик необязателен;
может работать реактивно)

Переносимость

- Реализация тредов на макросах и стейт машинах позволяет
  - Переносить на другой рантайм
  - Не зависеть от API тредов платформы
  - Самостоятельно управлять диспетчеризацией задач
- Реализация коммуникаций через каналы позволяет:
  - Не зависеть от асинхронных примитивов платформы (промисы, акторы, треды)
  - Явно организовать точки, в которых управление передается планировщику

Проблемы

```clojure
;; Clojure

(def ch (chan))

(defn my-async-task []
  (let [data (<! ch)] ;; compile error: использование <! вне go-треда
    (do-stuff data))

(go (my-async-task))
```

```go
// Go

var ch = make(chan int)

func myAsyncTask() {
    data := <-ch // нет проблем
    doStuff(data)
}

func main() { go myAsyncTask() }
```

```clojure
;; Fennel (мой порт core.async на корутинах)

(local ch (chan))

(fn my-async-task []
  (let [data (<! ch)] ;; нет проблем
    (do-stuff data))

(go (my-async-task))
```


Проблемы

```clojure
;; Clojure (стейт-машины)

(def ch1 (chan))
(def ch2 (chan))
(def ch3 (chan))

(go
  ;; compile error:
  ;; использование “синтаксиса” <! как функции
  (map <! [ch1 ch2 ch3]))

(go
  (map
   ;; compile error: нельзя трансформировать
   ;; анонимную функцию в стейт-машину
   (fn [c] (<! c))
   [ch1 ch2 ch3]))
```

```clojure
;; Fennel (корутины)

(local ch1 (chan))
(local ch2 (chan))
(local ch3 (chan))

(go
  ;; OK:
  ;; <! это функция
  (map <! [ch1 ch2 ch3]))

(go
  (map
   ;; OK:
   ;; go ничего не трансформирует
   (fn [c] (<! c))
   [ch1 ch2 ch3]))
```


Итого

- Плюсы:
  - Реализация на макросах позволяет переносить на любой диалект Clojure
  - Стейт-машины позволяют не зависеть от примитивов платформы для остановки и возобновления выполнения
  - Каналы позволяют нам четко определить точки остановки и выполнения
- Минусы:
  - Есть нюансы организации кода, отсутствующие при полноценной поддержке асинхронности в платформе
  - Необходимость писать “обёртки” для взаимодействия с другими системами асинхронного программирования

Спасибо!

Материалы к презентации

health-samurai.io

F [SCALA]

* ОЦЕНИТЕ ДОКЛАД