Concurrency und State in Clojure

Nicolai Mainiero, sidion GmbH

Bei funktionalen Programmiersprachen wird immer wieder hervorgehoben, dass sich nebenläufige (concurrent) Programme einfacher realisieren lassen. In Clojure wurde neben den effizienten unveränderlichen (immutable) Datenstrukturen transaktionaler Speicher (software transactional memory) implementiert, um den Zustand zu kontrollieren. Damit hat der Programmierer ein mächtiges Werkzeug, um koordiniert und synchron den Zustand der Anwendung zu ändern.

Wenn in einer Anwendung Nebenläufigkeit verwendet wird, ist es immer aufwendig, den Zustand während der Ausführung zu kontrollieren, um Fehler wie race conditions oder lost updates zu verhindern. Wenn die Anwendung auch noch auf einem System mit mehreren Prozessorkernen ausgeführt wird, vergrößert sich die Wahrscheinlichkeit, dass bei einer fehlerhaften Implementierung subtile Fehler auftreten, die schwer bis unmöglich zu debuggen sind. Threads, die Low-Level-Schnittstelle, um nebenläufige Programme zu modellieren, werden auch in Clojure genutzt, um nebenläufige Programme zu schreiben. Die Integration geht sogar so weit, dass jede Funktion das `java.util.concurrent.Callable`-Interface implementiert. Dadurch kann jede Clojure-Funktion mit der High-Level-Abstraktion von Java, den Executors, zusammen verwendet werden.

Identitäten und Werte

Clojure unterscheidet klar zwischen Identitäten (identity) und Werten (value), zum Beispiel könnte die deutsche Nationalmannschaft jede Spielerin in ihrem Kader tauschen und würde dennoch die deutsche Nationalmannschaft bleiben. Die Identität bleibt unverändert, ihr Wert ändert sich jedoch im Laufe der Zeit. Das bedeutet, wenn wir eine Identität – die deutsche Nationalmannschaft – ändern, definieren wir einen neuen Wert für einen bestimmten Zeitpunkt beziehungsweise Zeitraum. Es ist aber möglich, zum Kader von 2019 zurückzukehren, um herauszufinden, wer damals der Nationalmannschaft angehört hat.

In Clojure sind alle Datenstrukturen unveränderlich und persistent *[3]*. Dadurch kann man sie wie Werte behandeln und das macht sie automatisch Thread-safe, das bedeutet, sie können von verschiedenen Threads bearbeitet werden, ohne sich gegenseitig zu stören.

Für Situationen, in denen dieses Modell nicht geeignet ist, bietet Clojure folgende verschiedene Referenztypen (Identitäten):

- **Atoms** für unkoordinierte synchrone Änderungen an geteiltem Zustand
- **Refs** für koordinierte synchrone Änderungen an geteiltem Zustand
- **Agents** für asynchrone Änderungen an geteiltem Zustand
- **Vars**, um Thread-local Zustand zu handhaben

Diese Unterscheidung zwischen Wert und Identität wird in anderen Sprachen oft vermischt. Das erfordert dann von der Programmiererin, dass sie beim Schreiben von nebenläufigem Code besonders aufpassen muss, um keine Fehler zu machen. Diese Sprachen greifen oft auf Locks, Mutexe, Semaphoren oder das defensive Kopieren zurück, um eine gleichzeitige Veränderung einer Variablen zu verhindern. Das kann allerdings mit zunehmender Komplexität der Anwendung schnell zu Problemen wie Verklemmungen (deadlocks) oder race conditions führen. Clojure vereinfacht dies, da der überwiegende Teil der Codebasis funktional ist beziehungsweise frei von Seiteneffekten realisiert werden kann und die wenigen Teile, die von der Veränderbarkeit profitieren, klar ersichtlich sind, da sie einen der vier Referenztypen verwenden.

```
(def current-series (atom "Lucifer")) ;1
-> #'user/current-series
(deref current-series) ;2
-> "Lucifer"
@current-series ;3
-> "Lucifer"
(reset! current-series "The Expanse") ;4
-> "The Expanse"
```

Listing 1: Erzeugen und Ändern eines Atoms

Unkoordinierte synchrone Änderungen mit Atomen

Atome bieten den einfachsten Mechanismus, um Zugriff auf eine Datenstruktur zu steuern. Ein Atom wird durch folgende Funktion definiert: `(atom initial-state options?)`. In *Listing 1* sieht man, wie ein Atom erzeugt, gelesen (dereferenziert) und geändert werden kann.

In Zeile 1 wird unter dem Namen `current-series` ein neues Atom mit dem Wert „Lucifer" erzeugt. Ein Atom beinhaltet also immer einen Wert. Mit der `(deref ref)`- Funktion beziehungsweise mit dem @ Reader-Makro können das Atom dereferenziert und der aktuelle Wert gelesen werden. Mit `(reset! atom newval)` in Zeile 4 kann dem Atom ein neuer Wert zugewiesen werden. Dabei wird der alte Wert unkontrolliert überschrieben. Mithilfe der `(swap! atom f)`-Funktion wird der aktuelle Wert des Atoms durch die übergebene Funktion `f` aktualisiert. Das hat den Vorteil, dass die gewünschte Änderung auch mit dem tatsächlichen Wert des Atoms durchgeführt wird, auch wenn mehrere Threads gleichzeitig den Wert ändern wollen. Die Funktion `f` sollte keine Seiteneffekte haben, da sie eventuell mehrfach aufgerufen wird, bis die Aktualisierung erfolgreich war.

In Atomen kann man auch komplexere Datenstrukturen, wie zum Beispiel eine Map, speichern. Dadurch ist es möglich, zusammenhängende Daten atomar zu ändern, wie es in *Listing 2* demonstriert wird. Hier wurden nun der Titel der Serie und die Streaming-Plattform in einer Map zusammengefasst. Wenn man nun die Serie wechselt, so kann dies mithilfe von `swap!` und einer entsprechenden Funktion (`(assoc map key val)`) realisiert werden.

Refs und *transaktionaler Speicher (STM)*

Atome steuern den synchronisierten unkoordinierten Zugriff, es ist also nicht möglich, Änderungen über mehrere Atome koordiniert durchzuführen. Um dies zu ermöglichen, wurde in Clojure transaktionaler Speicher implementiert. Dazu wird eine Ref analog zu einem Atom erstellt, wie es in *Listing 3* zu sehen ist. Die Erzeugung und der Zugriff erfolgen auf Refs genauso wie auf Atome.

Das Ändern einer Ref unterscheidet sich allerdings von dem eines Atoms, wie in *Listing 4* zu sehen ist. Das STM sorgt dafür, dass eine Ref nicht ohne umschließende Transaktion geändert werden kann. Diese Transaktion kann mit der `(dosync & exps)`-Funktion erzeugt werden.

```
(def current-series (atom {:title "Lucifer" :platform "Prime"}))
(swap! current-series assoc :title "Star Trek: Lower Decks")
-> {:title " Star Trek: Lower Decks ", :platform "Prime"}
```

Listing 2: Ändern einer komplexen Datenstruktur

```
(def current-series (ref "The Expanse"))
-> #'user/current-series
(deref current-series)
-> "The Expanse"
@current-series  ; Alternative zur deref Funktion
-> "The Expanse"
```

Listing 3: Deklaration und Dereferenzierung einer Ref

Innerhalb des `dosync`-Blocks kann `ref-set` beliebig oft aufgerufen werden. Clojure garantiert, dass für solche Transaktionen folgende Eigenschaften gelten:

- atomar – wenn mehr als eine Ref in einer Transaktion geändert wird, wird die Änderung für einen Außenstehenden bei allen Refs gleichzeitig sichtbar
- konsistent – Refs können Validierungsfunktionen definieren, die, wenn eine davon fehlschlägt, dazu führen, dass die ganze Transaktion fehlschlägt
- isoliert – Transaktionen sehen keine Teilergebnisse von anderen Transaktionen

Da die Transaktionen bei Clojure im Speicher ablaufen, wird keine Dauerhaftigkeit wie bei einer Datenbank garantiert. *Listing 5* zeigt, wie mehrere Refs in einer Transaktion koordiniert geändert werden können.

In dem Beispiel wird der bestehende Wert einfach überschrieben, wie wir es auch schon bei den Atomen gesehen haben. Das nächste Beispiel in *Listing 6* beschreibt eine vereinfachte Chatanwendung, die aber ausreichend komplex ist, um zwei weitere Funktionen zum Aktualisieren von Refs zu demonstrieren.

Zunächst wird ein Record *[1]*, ein eigener Datentyp, der sich wie eine Map verhält, mit den Feldern `:sender` und `:text` definiert. Um alle bereits gesendeten Nachrichten zu speichern, wird eine Liste verwendet (Zeile 3). Das erste Element in dieser Liste ist dadurch automatisch die zuletzt gesendete Nachricht. Zeile 4 enthält eine naive Implementierung der `add-message`-Funktion. Erst wird die Ref dereferenziert, anschließend die neue Nachricht vorangestellt, um dann mit `ref-set` die Ref zu aktualisieren. Das ist sehr umständlich und nicht besonders funktional.

```
(ref-set current-series "Lucifer")
-> java.lang.IllegalStateException: No transaction running
(dosync (ref-set current-series "Lucifer"))
-> "Lucifer"
```

Listing 4: Änderung des Werts einer Ref ohne Transaktion

Wie bei den Atomen gibt es auch für Refs eine Funktion, um eine Datenstruktur innerhalb einer Ref zu ändern. Diese nennt sich passenderweise `(alter ref update-fn & args...)`. Dieser Funktion übergibt man eine Ref, eine Update-Funktion und gegebenenfalls Argumente für die Update-Funktion. In den meisten Fällen wird die `alter`-Funktion zum Ändern von Refs verwendet. In einigen Spezialfällen, wenn die Update-Funktion kommutativ (zum Beispiel a + b = b + a) ist, kann auf eine optimierte Funktion `(commute ref update-fn & args...)` zurückgegriffen werden. Diese erlaubt es Clojure, die Reihenfolge der Änderungen zu verändern und damit schneller auszuführen.

Um die Konsistenz zu gewährleisten, kann bei der Deklaration einer Ref eine Validierungsfunktion angegeben werden. *Listing 7* erweitert die Chatanwendung und fügt solch eine Validierungsfunktion hinzu, damit sichergestellt ist, dass jede Nachricht gültige Werte für `:sender` und `:text` hat.

Agents für asynchrone Änderungen

Für den Fall, dass eine Änderung am Zustand der Anwendung aufwendig ist oder auch erst später erfolgen kann, stellt Clojure mit Agents eine asynchrone Lösung bereit. Die Verwendung von Agents unterscheidet sich nicht groß von Atomen oder Refs. *Listing 8* zeigt, wie ein Agent erzeugt wird, wie er geändert wird und wie das Ergebnis abgeholt werden kann.

```
(def current-series (ref "Lucifer"))
-> #'user/current-series
(def current-stream-provider (ref "Prime"))
-> #'user/current-stream-provider
(dosync
  (ref-set current-series "Das Damengambit")
  (ref-set current-stream-provider "Netflix"))
-> "Netflix"
```

Listing 5: Koordinierte Änderungen mehrerer Refs

```
(defrecord Message [sender text]) ;1
-> user.Message
(->Message "Murtaugh" Ich bin zu alt für diesen Mist.") ;2
-> #user.Message{:sender "Murtaugh", :text "Ich bin zu alt für diesen Mist."}
(def messages (ref ())) ;3
-> #'user/messages
; bad idea
(defn naive-add-message [msg]
    (dosync (ref-set messages (cons msg @messages)))) ;4
-> #'user/naive-add-message
(defn add-message [msg]
    (dosync (alter messages conj msg))) ;5
-> #'user/add-message
(add-message (->Message "user 1" "hello")) ;6
-> (#user.Message{:sender "user 1", :text "hello"})
(add-message (->Message "user 2" "howdy")) ;7
-> (#user.Message{:sender "user 2", :text "howdy"}
    #user.Message{:sender "user 1", :text "hello"})
```

Listing 6: Einfache Chatanwendung

```
(defn valid-message? [msg]
(and (:sender msg) (:text msg)))
-> #'user/valid-message?
(def validate-message-list #(every? valid-message? %))
-> #'user/validate-message-list
(def messages (ref () :validator validate-message-list))
-> #'user/messages
(add-message "not a valid message")
-> java.lang.IllegalStateException: Invalid reference state
(add-message (->Message "Riggs" "Du bist der beste Cop den ich kenne."))
-> (#:user.Message{:sender " Riggs ", :text " Du bist der beste Cop den ich kenne."})
```

Listing 7: Validierung bei Ref-Änderungen

```
(def counter (agent 0))
-> #'user/counter
(send counter inc)
-> #object[clojure.lang.Agent 0x7be3c47d {:status :ready, :val 1}]
@counter
-> 1
```

Listing 8: Asynchrone Änderungen mit Agents

Der Funktionsaufruf `send` liefert im Gegensatz zu `swap!` oder `alter` nicht den aktualisierten Wert zurück, sondern eine Referenz auf den Agent. Die Dereferenzierung erfolgt jedoch genauso wie bei den Refs oder Atoms mit `deref` oder dem @ Reader-Makro.

Auch bei Agents kann, analog zu den Refs, eine Validierungsfunktion angegeben werden. Im Unterschied zu Refs, die synchron verarbeitet werden und bei denen ein Fehler bei der Validierung sofort bemerkt wird, muss bei der Verwendung von Agents überprüft werden, ob die Änderung erfolgreich war. *Listing 9* zeigt, wie man dabei vorgehen kann.

In Zeile 1 erzeugen wir einen Agent mit einer Validierungsfunktion, die prüft, ob der neue Wert, den wir im Agent speichern wollen, auch eine Zahl ist. Dann senden wir eine Updatefunktion an den Agent, die immer einen String zurückliefert, die Validierung also absichtlich verletzt (Zeile 2). Wenn diese Funktion schlussendlich ausgeführt wird, führt es zu einer Ausnahme im Agent und je nach Fehlermodus `:fail` oder `:continue` verhält sich der Agent unterschiedlich. Wurde beim Erzeugen des Agent kein `:error-handler` angegeben, befindet er sich im `:fail`-Modus. Jede Ausnahme führt dazu, dass der Agent in einen Ausnahmezustand wechselt. Diesen Zustand kann der Agent nur verlassen, wenn er neu gestartet wird (Zeile 4).

```
(def counter (agent 0 :validator number?)) ;1
-> #'user/counter (send counter inc)
(send counter (fn [_] "boo")) ;2
-> #object[clojure.lang.Agent 0x9ee5b57c {:status :ready, :val 0}]
(agent-error counter) ;3
-> #error {
 :cause "Invalid reference state"
 :via
 [{:type java.lang.IllegalStateException
   :message "Invalid reference state"
   :at [clojure.lang.ARef validate "ARef.java" 33]}]
 :trace
 […]}
(restart-agent counter 0) ;4
-> 0
@counter
-> 0
(defn handler [agent err]
  (println "ERR!" (.getMessage err))) ;5
-> #'user/handler
(def counter2
  (agent 0 :validator number? :error-handler handler)) ;6
-> #'user/counter2
(send counter2 (fn [_] "boo")) ;7
-> #object[clojure.lang.Agent 0xbae653d9 {:status :ready, :val 0}]
user=> ERR! Invalid reference state
(send counter2 inc) ;8
-> #object[clojure.lang.Agent 0xbae653d9 {:status :ready, :val 0}]
@counter2 ; 9
-> 1
```

Listing 9: Validierung bei Agents

```
(def backup-agent (agent "output/messages-backup.clj")) ;1
-> #'user/backup-agent
(defn add-message-with-backup [msg]
  (dosync
    (let [snapshot (commute messages conj msg)]
    (send-off backup-agent (fn [filename]
                             (spit filename snapshot)
                              filename))
snapshot))) ;2
-> #'user/add-message-with-backup
(add-message-with-backup (->Message "John" "Message One")) ;3
-> (#user.Message{:sender "John", :text "Message One"})
```

Listing 10: Kombination von Transaktionen und Agents

Wurde beim Erstellen des Agent ein `:error-handler` angegeben, befindet sich der Agent im `:continue`-Modus. In diesem Modus wird im Fehlerfall der übergebene `error-handler` aufgerufen (Zeile 7). Der Agent bleibt aber ganz normal benutzbar (Zeile 8 und 9).

Agents und Transaktionen

Die Funktionen zum Ändern von Atomen oder Refs sollten frei von Seiteneffekten sein, da Clojure eventuell versucht, die Aktualisierungsfunktion mehrfach aufzurufen, bis die Aktualisierung erfolgreich war. Allerdings kann es vorkommen, dass bei einer erfolgreichen Transaktion ein Seiteneffekt ausgelöst werden soll. Hier kann auf Agents zurückgegriffen werden. Wenn einem Agent innerhalb einer Transaktion eine Anweisung geschickt wird, wird diese genau einmal und auch nur bei Erfolg der Transaktion ausgeführt. Ergänzen wir die Chatanwendung um ein Backup, das die Nachrichten in eine Datei schreibt *(siehe Listing 10)*.

Die neue Funktion `add-message-with-backup` in Zeile 2 erweitert die alte Funktion und speichert zunächst die aktualisierte Liste der Nachrichten in der Variablen `snapshot`, um dann mit `send-off` an den Agent die Anweisung zur Aktualisierung der Datei zu senden. Im Unterschied zu `send`, das wir bereits kennengelernt haben, wird hier `send-off` verwendet. Diese Funktion wird immer dann verwendet, wenn es sich um einen blockierenden Vorgang, wie das Schreiben in einer Datei, handelt. Es ist verlockend, diese Konstruktion als Datenbank zu verwenden, Clojure STM liefert ACI und das Schreiben in die Datei trägt das D von ACID bei. Allerdings garantiert Clojure nicht, dass der Agent innerhalb der ACI-Transaktion ausgeführt wird. Das bedeutet, die Transaktion kann erfolgreich gewesen sein und der Agent versucht nun, auf das Dateisystem zuzugreifen. Wenn das jetzt fehlschlägt, geht die Anwendung davon aus, dass die Daten persistiert worden sind, auch wenn das gar nicht der Fall ist.

Einheitliches Aktualisierungs-Modell

Clojure legt sehr viel Wert auf konsistente APIs. Das bemerkt man auch beim Umgang mit Atomen, Refs und Agents. Sie werden alle ähnlich erzeugt und gelesen und können auf die gleiche Art und Weise geändert werden. Hier nochmals zusammengefasst, welche Methode wofür verwendet wird *(siehe Tabelle 1)*.

Fazit

Clojure bietet sehr flexible und ausgefeilte Möglichkeiten, um den Zustand einer Anwendung über mehrere Threads hinweg zu managen. Besonders hervorzuheben ist das einheitliche Modell, um mit Atomen, Refs oder Agents zu interagieren. Durch die unveränderlichen Datenstrukturen und STM ermöglicht es, Aktualisierungen über Threads hinweg zu koordinieren. Und wenn die Threads unabhängig voneinander sind, muss man sich keine Gedanken machen, ob ein Dritter die eigenen Daten gerade geändert hat.

	Atom	Ref	Agent
Änderungsfunktion	`alter`	`swap!`	`send-off`
Änderungsfunktion (kommutativ)	`commute`	-	-
Änderungsfunktion (non-blocking)	-	-	`send`
Setter	`reset!`	`ref-set`	-

Tabelle 1

Quellen

[1] Alex Miller (2018): Programming Clojure. The Pragmatic Bookshelf, Raleigh, North Carolina

[2] Michael Sperber (2015): Zusammengesetzte Daten in Clojure, *https://funktionale-programmierung.de/2015/04/27/clojure-records.html*

[3] Clojure Data Structures, *https://clojure.org/reference/data_structures*

Nicolai Mainiero
sidion GmbH
nicolai.mainiero@sidion.de

Nicolai Mainiero ist Diplom-Informatiker und arbeitet als Software Developer bei der sidion GmbH. Er entwickelt seit über 13 Jahren Geschäftsanwendungen in Java, Kotlin und Clojure für unterschiedlichste Kundenprojekte. Dabei setzt er vor allem auf agile Methoden wie Kanban. Außerdem interessiert er sich für funktionale Programmierung, Microservices und reaktive Anwendungen.