Funktional, parallel, genial

Einführung in Clojure – Teil 2

Burkhard Neppert, Stefan Tilkov

Clojure kombiniert das Beste aus Lisp mit einer perfekten Java-Integration. Im ersten Teil der Clojure-Einführung haben Sie die Syntax, Funktionen höherer Ordnung und einige Kontrollstrukturen kennengelernt. In diesem Teil soll es um Clojures strukturierte Datentypen und die Integration mit Java gehen.

Weitere Datentypen

Neben den einfachen Datentypen wie Zahlen, Strings und Booleans sind in Clojures Sprachkern Listen, Vektoren, Mengen und Maps (assoziative Arrays) vorhanden.

Listen

Eine Liste ist eine geordnete Abfolge von Clojure-Werten. Eine Syntax, mit der Listen erzeugt werden, wurde bereits im ersten Teil [NeTi10] kurz gezeigt: `'(<ausdruck>...)` erzeugt eine Liste, die die nicht ausgewerteten Ausdrücke als Elemente enthält. Beispiel:

```
>> '(1 2 3) ==> (1 2 3)
>> '(1 2 (+ 1 2)) ==> (1 2 (+ 1 2))
```

Sollen die Ausdrücke in ausgewerteter Form als Listenelement erscheinen, muss die Funktion `list` verwendet werden:

```
>> (list 1 2 (+ 1 2)) ==> (1 2 3)
```

Die Elemente von Listen können beliebige Clojure-Werte sein, also auch andere Listen. Auf diese Weise können komplexere Datenstrukturen konstruiert werden.

Was kann man mit einer Liste anfangen?

- Elemente zählen:

```
>> (count '(1 2 3)) ==> 3
```

- Auf das erste Element zugreifen:

```
>> (first '(1 2 3)) ==> 1
```

- Die Liste aller Elemente außer dem ersten zurückgeben lassen:

```
>> (rest '(1 2 3)) ==> (2 3)
```

- Zugriff auf das n-te Element:

```
>> (nth '(0 1 2 3 4) 3) ==> 3
```

- Eine neue Liste mit einem zusätzlichen Element am Beginn erzeugen:

```
>> (conj '(1 2 3) 0) ==> (0 1 2 3)
```

Das sieht sehr unspektakulär aus, ist aber zusammen mit Funktionen, die Funktionen kombinieren, erstaunlich mächtig. In Clojure werden im Sprachkern einige Funktionen bereitgestellt, die eine Funktion auf alle Elemente einer Liste anwenden und daraus neue Listen erzeugen.

Eine dieser Funktionen ist `map`. Sie wendet eine Funktion auf jedes Element einer Liste an und liefert eine neue Liste mit den Funktionsergebnissen als Wert:

```
>> (map (fn[x] (* x x)) '(1 2 3 4 5)) ==> (1 4 9 16 25)
```

Die Funktion `reduce` erwartet als erstes Argument eine Funktion `f` mit zwei Parametern und als zweites Argument eine Liste von Werten. `f` wird auf die ersten beiden Elemente der Liste angewendet, dann wird `f` auf dieses Ergebnis und das dritte Element der Liste angewendet, und so weiter:

```
>> (reduce + '(1 2 3 4 5)) ==> (+ (+ (+ (+ 1 2) 3) 4) 5) ==> 15

>> (defn rückwärts[lst]
     (reduce conj '() lst))

>> (rückwärts [1 2 3 4])
==> (conj 4 (conj 3 (conj 2 (conj 1  '()))))  ==> (4 3 2 1)
```

Eine Besonderheit von Clojure-Listen soll hier noch erwähnt werden: Clojure kennt auch sogenannte „lazy sequences". Das sind Listen von berechneten Werten, deren Berechnung so lange verzögert wird, bis tatsächlich auf ein Element der Liste zugegriffen wird. Eine Funktion, die „lazy sequences" erzeugt, ist `iterate`. `(iterate <funktion> <startwert>)` generiert eine unendliche Liste durch wiederholte Anwendung von `<funktion>`. Ein schönes Beispiel ist die Liste aller natürlichen Zahlen. Diese lässt sich in Clojure mit `iterate` wie folgt definieren:

```
>> (def n0 (iterate inc 0))
==> ;; (0 (inc 0) (inc (inc 0)) …)
==> (0 1 2 3 4 5 6 …)
```

Wie aber kann man mit einer solchen Liste etwas anfangen? Würde man sie zum Beispiel auf die Konsole ausgeben, könnte man lange warten. Der Trick besteht darin, dass irgendwann die Evaluation nur für einen Teil der potenziell unendlichen Ergebnismenge angestoßen wird. Ein Beispiel dafür ist die Funktion `take`:

```
>> (take 5 (iterate inc 0)) ==> (0 1 2 3 4)
```

„Lazyness" funktioniert natürlich nicht nur für unendliche Listen, sondern auch für solche, die einfach nur sehr groß oder aufwendig zu evaluieren sind. Darüber hinaus lassen sich Funktionen, die lazy sind, definieren und miteinander kombinieren. Die Funktion `map` erzeugt als Ergebnis eine „lazy sequence":

```
>>  (def lazy (map (fn [x]
                     (println x "*"  x "=" (* x x))
                     (* x x))
                   (range 1 100)))  ;; Keine Ausgabe
>> (nth lazy 10)   ;; Die ersten Elemente werden berechnet
```

Wenn bei der Berechnung von `map` Seiteneffekte erzeugt werden sollen oder die Berechnung schon zum Zeitpunkt des `map`-Aufrufs erfolgen soll, muss die Berechnung mit `doall` erzwungen werden:

```
>>  (def eager (doall
                  (map (fn [x]
                         (println x "*"  x "=" (* x x))
                         (* x x))
                       (range 1 100))))  ;; Alle 100 Ausgaben sofort
```

Vektoren

Vektoren sind wie Listen eine geordnete Menge von Clojure-Werten. Vektoren werden durch `[<elment> ...]` bzw. `(vector <element> ...)` erzeugt. Auf Vektoren können die gleichen Operationen wie auf Listen angewendet werden, auch `map`, `reduce` usw. Zudem kann auf ein beliebiges Element im Vektor in beinahe konstanter Zeit, genauer in O(log32(n)), zugegriffen werden.

Der Elementzugriff kann entweder durch `(nth <array> <index>)` oder `(<array> <index>)` erfolgen. Ein neuer Vektor kann aus einem existierenden durch `(assoc <vektor> <index> <neuer-wert>)` erzeugt werden. Der neue Vektor hat an der Position `<index>` den Wert `<neuer-wert>`:

```
>> (def abc [:a :b :c])
>> (assoc abc 0 :eins) ==> [:eins :b :c]
>> (nth abc 0) ==> :a
```

Der alte Vektor bleibt dabei unverändert.

Maps

Assoziative Arrays bzw. Maps bilden Schlüssel-Wert-Paare ab. Das folgende Beispiel zeigt eine Map, in der Namen Adressen zugeordnet werden. Als Schlüssel werden dabei Keywords verwendet, die Werte sind wiederum Maps:

```
>> (def people { :bn {:ort "Nürnberg", :plz "90321"},
                 :st {:ort "Ratingen", :plz "40880"} })
```

Eine Map lässt sich wie eine Funktion verwenden, schließlich liefert sie für einen Schlüssel immer den gleichen Wert zurück:

```
>> (people :st) ==> {:ort "Ratingen", :plz "40880"}
```

Mit **(assoc <map> <schlüssel> <wert>)** wird eine neue Map erzeugt, die für **<schlüssel>** den Wert **<wert>** enthält:

```
>> (assoc people :pg {:ort "Ratingen" :plz "40880"})
==>  {:pg {:ort "Ratingen", :plz "40880"},
      :bn {:ort "Nürnberg", :plz "90321"},
      :st {:ort "Ratingen", :plz "40880"}}
```

StructMaps

Maps sind auch die Grundlage für Clojures *StructMap*. Eine StructMap ist ein assoziatives Array mit vorbelegten Schlüsseln. Mit **(defstruct <struct-name> <key>+)** definieren Sie eine StructMap, mit **(struct <struct-name> <key-value>*)** erzeugen Sie eine neue Instanz. Dabei werden die Werte **<key-value>** der Reihe nach den Schlüsselfeldern zugeordnet:

```
>> (defstruct Punkt2D :x :y)
>> (def p (struct Punkt2D 1 2))
>> (p :x) ==> 1
>> (p :y) ==> 2
```

Integration mit Java

Nutzung von Java-Objekten in Clojure-Funktionen

Clojure ist als Sprache für die JVM konzipiert und versteckt die darunterliegende Infrastruktur nicht. Clojures elementare Datentypen wie Zahlen oder Zeichenketten sind die entsprechenden Java-Klassen (**java.lang.Integer**, **java.lang.Long**, ...). Java-Objekte können in Clojure-Programmen durch **(new <Klassenname> <Argumente>*)** erzeugt werden. Alternativ ist noch die Syntax **(<Klassenname>. <Argumente>*>)** implementiert:

```
>> (new Integer 42) ==> 42
>> (Integer. 42 ) ==>  42
```

Methoden von Instanzen werden mit **(.<Methodenname> <Java-Objekt> <Argumente>*)** aufgerufen:

```
>> (.toLowerCase "ABC") ==> "abc"
```

Statische Methoden werden durch **(<Klassenname>/<Methodenname> <Argumente>*)** aufgerufen, auf statische Felder wird mit **<Klassenname>/Feldname** zugegriffen:

```
>> (Math/cos  Math/PI) ==> -1.0
```

Java-Klassen außerhalb des Pakets java.lang müssen standardmäßig im Clojure-Code mit ihrem voll qualifizierten Namen angesprochen werden. Mit einem **(:import [<paket> <klasse>+])** in der Clojure-Namensraumdeklaration werden die Klassen innerhalb des Namensraums durch ihren unqualifizierten Namen zugänglich:

```
>> (new java.util.Date)
>> (ns kürzer (:import [java.util Date ArrayList]))
>> (new Date)
```

Viele der eingebauten Clojure-Funktionen arbeiten mit Java-Objekten als Argument. So können alle Java-Klassen, die **java.lang.Iterable** oder **java.util.Map** implementieren, sowie Strings und Java-Arrays als Clojure-Sequenzen behandelt werden. Clojure-Funktionen, die eine Sequenz als Argument erwarten, können dann mit diesen Java-Klassen verwendet werden.

Implementierung von Java-Interfaces mit Clojure

Clojure kann existierende Java-Klassen und Interfaces durch Proxy-Objekte implementieren. Proxy-Objekte sind Instanzen, die von einer Java-Klasse erben und eine Reihe von Interfaces implementieren. Proxy-Objekte werden durch die Clojure-Funktion **proxy** erzeugt. Diese erhält als erstes Argument einen Vektor, bestehend aus (maximal) einer Java-Klasse, gefolgt von beliebig vielen Java-Interfaces. Das zweite Element ist ein Vektor, der die Konstruktor-Argumente der Basisklasse des Proxys enthält. Danach folgen die Funktionsdefinitionen des Interfaces als Listen **(<funktions-name> [<argument>*] <ausdruck>*)**.

Zur Demonstration schreiben wir eine Funktion **new-action-listener**. Diese Funktion erzeugt Proxy-Objekte, die das **java.awt.event.ActionListener**-Interface implementieren. Als Argument erhält **new-action-listener** eine Funktion **handler-fn**, die in der Implementierung von **ActionListener.actionPerformed** benutzt wird, Rückgabewert ist das Proxy-Objekt:

```
>> (defn new-action-listener
    [handler-fn]
     (proxy [java.awt.event.ActionListener]
        ;; Das implementierte Interface
            []  ;; Leere Argumentliste für Basisklassen-Konstruktor
       (actionPerformed [event]
          ;; Die Implementierung des Interface:
          ;; wende handler-fn auf das Event-Objekt an.
          (handler-fn event))))
```

Als konkrete Nutzung nun ein rudimentäres „Whack-a-mole"-Spiel: Der Spieler versucht so schnell wie möglich ein Objekt mit der Maus zu treffen. Hier ist das Objekt ein JButton und das Verhalten bei einem Treffer ist, an zufälliger Stelle neu zu erscheinen. Das Verhalten wird in der Funktion **random-jump** implementiert:

```
>> (let [randgen (java.util.Random.)]
     ;; Durch "let [randgen ...] wird ein Zufallgenerator erzeugt,
     ;; der nur lokal in der folgenden Funktionsdefinition
     ;; sichtbar ist.
     (defn random-jump [component]
         (let [parent-size (.getSize (.getParent component))]
           (.setLocation component   ;;  Position im Parent
                                     ;; zufällig setzen
                   (.nextInt randgen (.getWidth parent-size))
                   (.nextInt randgen (.getHeight parent-size))))))
```

Die Elemente der GUI werden in der Funktion **whack-a-button** erzeugt und zusammengesetzt. Dem Zielobjekt **button** wird als ActionListener das mit **new-action-listener** erzeugte Proxy-Objekt zugewiesen:

```
>> (defn whack-a-button [width height btn-size]
    (let [frame  (JFrame. "Whack-a-Button")
          button (JButton. "")]
      (.setBounds button 10 10 btn-size btn-size)
      (.addActionListener button
                    (new-action-listener (fn [event]
```

```
                                        (random-jump button))))
      (doto frame                ;; (doto frame (. ....) )
                                 ;; ist das Gleiche wie
        (.setLayout nil)         ;; (.setLayout frame nil)
        (.setSize width height)  ;; (.setSize frame width height
        (.add button)            ;; (.add frame button)
        (.setVisible true))))    ;; (.setVisible frame true)
                                 ;; frame

>> (whack-a-button 600 600 30)
==> Treffen Sie den Button?
```

Das Beispiel verwendet die Clojure-Syntax `doto`, um nacheinander mehrere Methoden an einem Java-Objekt aufzurufen.

Nutzung von Clojure-Funktionen in Java-Programmen

Soll ein Java-Programm Clojure-Code verwenden, kann die Laufzeitklasse `clojure.lang.RT` des Clojure-Systems verwendet werden. `clojure.lang.RT` ist der Einstiegspunkt zu den Interna der Clojure-Implementierung. Alle Methoden der Clojure-Laufzeitumgebung sind als statische Methoden implementiert, die auf einer gemeinsamen Umgebung arbeiten. Pro Instanz einer Java-VM gibt es nur eine Clojure-Instanz.

Das folgende Java-Programm zeigt, wie Clojure-Code aus Java aufgerufen wird. Zunächst wird mit der Methode `loadResourceScript` Clojure-Code aus dem Klassenpfad der VM gelesen und in Bytecode übersetzt. Das Java-Objekt, das eine Definition im Clojure-Code repräsentiert, wird dann mit `RT.var(<Namespace>, <Name>)` von der Clojure-Laufzeitumgebung zurückgegeben:

```
/** In der Datei com/innoq/clj/hello.clj:
  (ns com.innoj.clj)
  (defn hallo [arg] (println "Hallo" arg))
*/

package com.innoq.clj;
import clojure.lang.RT;
import clojure.lang.Var;
import clojure.lang.IFn;

public class RTSample {
  public static void main(String[] args) {
   try {
     RT.loadResourceScript("com/innoq/clj/hello.clj");
     Var hallo=RT.var("com.innoq.clj", "hallo");
     if(hallo.isBound()                  // Ist ein Wert definiert?
        && hallo.deref() instanceof IFn) // und ist dieser Wert
                                         // eine Funktion?
     {
       Object result=hallo.invoke("lieber Leser");
       if(result!=null) System.out.println("Rückgabewert: "+result);
     } else {
       System.out.println(
         "Funktion com.innoq.clj/hallo nicht definiert.");
     }
   } catch(Exception e) {
       e.printStackTrace();
     }
   }
 }
```

Um die Suche nach Fehlern im Skript zu erleichtern, werden im Java-Code die Objekte der Clojure-Laufzeit genauer untersucht. Clojures Bindungen von Symbolen zu Werten werden durch die Klasse `Var` implementiert. Durch `Var.isBound()` wird getestet, ob ein Wert zugeordnet ist. Mit `Var.deref()` wird dieser Wert abgefragt und mit „`instanceof IFn`“ überprüft, ob es sich um eine Clojure-Funktion handelt.

Diese Art der Integration greift direkt auf die Implementierungsdetails von Clojure zu. Sie sollte daher nur sparsam verwendet werden, z. B. wenn Clojure als Erweiterungssprache einer Anwendung genutzt werden soll.

Ausblick

In diesem Teil haben Sie Clojures eingebaute Datenstrukturen kennengelernt sowie einige der Möglichkeiten, Clojure zusammen mit Java zu nutzen. Im abschließenden Teil der Clojure-Einführung werden Sie Möglichkeiten für die Programmierung parallelen Codes mit Clojure kennenlernen.

Links

[clojure1.1.0] Clojure-Quellen und jar-Archiv, `http://clojure.googlecode.com/files/clojure-1.1.0.zip`

[NeTi10] B. Neppert, St. Tilkov, Clojure: Funktional, parallel, genial – Teil 1: Überblick, in: JavaSPEKTRUM, 2/2010

Burkhard Neppert ist Senior Consultant bei der innoQ Deutschland GmbH. Er beschäftigt sich dort mit Konzeption und Entwicklung unternehmenskritscher Systeme, vorwiegend in Mainstream-Sprachen wie Java oder C++.
E-Mail: burkhard.neppert@innoq.com.

Stefan Tilkov ist Geschäftsführer und Principal Consultant bei der innoQ Deutschland GmbH, wo er sich mit Architekturen für verteilte Systemlandschaften beschäftigt. Sein aktuelles Hauptinteresse liegt auf dynamischen Sprachen und dem Einsatz von RESTful HTTP für Integrationsarchitekturen.
E-Mail: stefan.tilkov@innoq.com.