Einfach Lisp

Clojure: Funktional, parallel, genial – Teil 1: Überblick

Burkhard Neppert, Stefan Tilkov

Clojure ist eine neue, fortgeschrittene, JVM-basierte Sprache, die das Beste aus Lisp mit einer modernen Umgebung, der Unterstützung für parallele Verarbeitung in Mehrkern-Umgebungen und perfekter Java-Integration kombiniert. Der Artikel gibt einen Überblick über die Grundkonzepte der Sprache und die Möglichkeiten zum praktischen Einsatz.

Lisp ist fünfzig Jahre alt – wer hätte gedacht, dass eine Sprache mit einem solchen Alter noch einmal eine Renaissance erleben könnte? Wenn Sie mit einem Lisp-Verfechter sprechen, wird dieser darauf antworten, dass eine solche Wiederbelebung gar nicht nötig ist und sich die Sprache bester Gesundheit erfreut. Das stimmt zum Teil, und es ist auch nicht von der Hand zu weisen, dass viele der Konzepte, die heute Eingang in die aktuellen Mainstream-Sprachen finden, schon seit Jahrzehnten im Lisp-Umfeld bekannt sind und genutzt werden. Seit langem scheitert die Akzeptanz von Lisp in der Praxis weniger an der etwas gewöhnungsbedürftigen Syntax als an der Vielzahl von Dialekten und der überaus mäßigen Ausstattung an Bibliotheken.

Der Lisp-Dialekt Clojure ist daher etwas Besonderes: Mit der JVM als Ablaufumgebung, einer perfekten Java-Integration und außerordentlich durchdachten Mitteln für die Parallelverarbeitung in Mehrkern-Umgebungen kombiniert Spracherfinder Rich Hickey die Stärken einer funktionalen Sprache mit einer weit verbreiteten Plattform. Anders ausgedrückt: Ein Lisp, das man tatsächlich praktisch einsetzen kann!

In dieser dreiteiligen Artikelserie möchten wir Ihnen einen Überblick über die Sprache Clojure geben:

- Der erste Teil wird sich mit den grundlegenden Dingen der Syntax und der funktionalen Programmierung beschäftigen.
- Der zweite Teil ist Clojures Datenstrukturen und dem Zusammenspiel von Clojure mit Java gewidmet.
- Den Abschluss bildet im dritten Teil eine Einführung in die Programmierung paralleler Anwendungen mit Clojure.

Probieren geht über Studieren

Wie andere dynamische Sprachen unterstützt auch Clojure einen „explorativen" Programmierstil: Durch die Read/Evaluate/Print-Loop (REPL) können Clojure-Funktionen interaktiv definiert und getestet werden. Dieses Vorgehen ersetzt zwar kein sorgfältiges Design und Testen, macht aber durch die unmittelbar erzielten Resultate das Ausprobieren von Ideen möglich (sozusagen Extreme Rapid Prototyping).

Wenn Clojure für Sie der erste Kontakt mit einer Lisp-ähnlichen Sprache ist, hilft das interaktive Herumexperimentieren in der REPL die Einstiegshürde leichter zu nehmen. Die REPL liest Ausdrücke der Programmiersprache, berechnet deren Wert und gibt die Werte aus. Durch die Berechnung des Wertes können interessante Dinge geschehen: Benutzeroberflächen werden angezeigt, Grafiken gemalt, Dateien geschrieben ...

In den folgenden Beispielen verwenden wir die REPL, um Clojure vorzustellen. Dabei steht „>>" für den REPL-Prompt, mit „==>" wird das Ergebnis der Auswertung notiert. Sie können die Beispiele somit als Startpunkt für eigene Experimente nutzen.

Der Installationsaufwand dafür ist minimal: [clojure1.1.0] herunterladen, entpacken und in einer Shell

```
java -jar <Pfad zu entpacktem zip>/clojure-1.1.0/clojure.jar
```

starten. Wer Clojure in einer Entwicklungsumgebung nutzen möchte, kann das Plug-In counterclockwise für Eclipse [counterclockwise] verwenden. Für weitere Informationen zur Einrichtung und Unterstützung für andere Entwicklungsumgebungen sei auf [clojure] verwiesen.

Was sind Clojure-Ausdrücke?

Clojure versteht konstante Ausdrücke für elementare Datentypen wie Zahlen, Zeichen, Zeichenketten oder Booleans (mit den Werten „true" und „false"). Bei der Auswertung von konstanten Ausdrücken werden diese unverändert als Wert zurückgegeben:

```
>> 1 ==> 1
>> 1.0 ==> 1.0
>> \c ==> \c
>> true ==> true
>> "Ein String" ==> "Ein String"
```

Eine besondere Art von konstanten Ausdrücken sind sogenannte Schlüsselwörter. Diese sind symbolische Bezeichner, die einen effizienten Test auf Gleichheit erlauben. Sie werden hauptsächlich als Schlüssel z. B. in Hashmaps eingesetzt. Ein Bezeichner, der mit einem Doppelpunkt beginnt, ist ein Schlüsselwort:

```
>> :schluessel ==> :schluessel
```

Das Pendant zu Javas `null` heißt in Clojure `nil`.

Als weitere Art von Ausdrücken gibt es Funktionsaufrufe. Ein Funktionsaufruf hat die Form einer Liste: `(<ausdruck> <ausdruck> ...)`*. Die Clojure-Laufzeitumgebung wertet zuerst alle Ausdrücke in der Liste aus. Der erste Ausdruck muss eine

* Aber nicht jede dieser Formen ist auch ein Funktionsaufruf. Es gibt Sonderformen wie „if", die anders als Funktionen nicht immer alle Argumente auswerten.

Funktion als Wert liefern. Diese Funktion wird mit den Werten der anderen Ausdrücke als Parameter berechnet und als Wert des Funktionsaufrufs zurückgegeben:

```
>> (+ 1 2 3) ; Kommentare beginnen mit „;“ und enden mit der Zeile
==> 6
```

Der erste Ausdruck, `+`, wird vom Laufzeitsystem zur eingebauten Additionsfunktion ausgewertet. Die anderen Ausdrücke, `1`, `2` und `3`, sind Zahlen-Konstanten. Es wird also die Addition mit `1`, `2` und `3` als Parameter aufgerufen und `6` als Wert zurückgegeben.

Funktionsaufrufe können auch beliebig geschachtelt werden:

```
>> (+ 1 (* 2 3))
==> (+ 1 6) ==> 7
```

Und schließlich gibt es noch die Möglichkeit, die Auswertung eines Ausdrucks durch ein vorangestelltes Hochkomma zu unterbinden (zu „quoten“):

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

Im Beispiel oben ist das Ergebnis eine Liste mit den Elementen `'+`, `1` und der Liste `'(* 2 3)`.

Clojure besitzt eine große Anzahl eingebauter Funktionen [clojureAPI], aber bis jetzt war Clojure nichts weiter als eine Art Taschenrechner mit einer Präfix-Syntax. In Clojure können aber auch eigene Funktionen definiert werden.

Funktionen bauen

Der Ausdruck, mit dem eine Funktion erzeugt wird, ist `fn`:

```
>> (fn [x y z]
     (+ x (* y z)))
```

Dieser Ausdruck erzeugt eine Funktion mit drei Parametern, `x`, `y` und `z`, die als Wert die Summe von `x` und dem Produkt aus `y` und `z` berechnet. Also:

```
>> ((fn [x y z] (+ x (* y z))) 1 2 3)
==> 7
```

Clojure ist eine Sprache mit dynamischer Typisierung. Der Typ eines Symbols ergibt sich aus dem Wert, der dem Symbol zugeordnet ist. In der oben definierten Funktion müssen die Typen der Parameter `x`, `y` und `z` sowie der Typ des Rückgabewertes nicht deklariert werden. Wird die Funktion auf Werte angewendet, die für die im Funktionsrumpf verwendeten Funktionen nicht zulässig sind, wird die Auswertung ohne Rückgabewert mit einer Exception abgebrochen:

```
>> ((fn [x y z] (+ x (* y z)))  :eins :zwei :drei)
==> java.lang.ClassCastException: clojure.lang.Keyword cannot be
cast to java.lang.Number)
```

Immer die Funktionsdefinition hinschreiben zu müssen, ist sicher nicht praktikabel, und so gibt es mit `(defn <symbol> [<argument> ... ] <ausdruck> ...)` die Möglichkeit, die Funktion einem Symbol zuzuordnen:

```
>> (defn plus-mal [x y z]
     (+ x (* y z)))
>> (plus-mal 1 2 3)
==> 7
```

Neben der Definition von Funktionen mit einer festen Parameterzahl können in Clojure auch Funktionen mit verschiedenen (aber festen) und variablen Argumentzahlen definiert werden. Bei einer variablen Argumentzahl muss als letzter Teil der Parameterliste `& <symbol>` stehen. Die restlichen Argumente sind dann innerhalb einer Liste `<symbol>` im Funktionsrumpf nutzbar:

```
>> (defn überladen
     ([] 0)           ;; Definition von „überladen“ ohne Argument,
     ([x] 1)          ;; mit einem Argument
     ([x y] 2)        ;; mit zwei Argumenten
     ([x y & rest]    ;; und mit mindestens drei Argumenten
      (+ 2 (count rest)))) ;; „count“ zählt die Anzahl der
                           ;; Listenelemente
>> (überladen) ==> 0
>> (überladen :a :b) –> 2
>> (überladen :a :b :c :d :e) ==> 5
```

Neben konstanten Ausdrücken, Funktionsaufrufen und Funktionsdefinitionen gibt es besondere syntaktische Formen. Diese sehen aus wie Funktionsaufrufe, werden aber von Clojure anders ausgewertet.

Bedingte Auswertung mit if

Der folgende Ausdruck

```
(if <test> <true-ausdruck> <else-ausdruck>?)
```

wertet den `<test>`-Ausdruck aus. Ist das Ergebnis weder `false` noch `nil`, so wird `<true-ausdruck>` ausgewertet und als Ergebnis des `if`-Ausdrucks zurückgegeben. Andernfalls wird entweder der Wert des optionalen `<else-ausdruck>` zurückgegeben oder, falls dieser fehlt, `nil`:

```
>> (if (> 1 2) :was? :ok) ==> :ok
>> (if nil :x) ==> nil
```

Sequenzielle Auswertung mit do

`(do <ausdruck> ...)` wertet alle Ausdrücke sequenziell aus und gibt den Wert des letzten Ausdrucks als Wert des `do` zurück. Die `do`-Form wird vor allem dann verwendet, wenn Sie Seiteneffekte, z. B. Ausgaben, erzeugen wollen:

```
>> (do
     (print :a)
     (println :b)) ==> nil
```

Symbol/Wert-Bindungen mit let

Bei der Definition von Funktionen kann der Fall auftreten, dass der Funktionsausdruck unhandlich groß wird oder Teilergebnisse an verschiedenen Stellen verwendet werden. Für diesen Fall existiert die Sonderform `let`:

```
(let [<symbol> <bindungs-ausdruck> ... ] <rumpf-ausdruck> ...)
```

`let` erzeugt eine „Umgebung“, also eine Verknüpfung von Symbolen mit Werten. Dazu wertet `let` die Bindungs-Ausdrücke in [...] aus und bindet sie an die Symbole. Die Auswertung der Werte für Bindungen geschieht in der deklarierten Reihenfolge. Symbole, die bereits gebunden wurden, können in den nachfolgenden Bindungsausdrücken verwendet werden.

Die Rumpfausdrücke werden danach sequenziell abgearbeitet und der Wert des letzten Ausdrucks als Wert des `let`-Ausdrucks zurückgegeben. Wird in den Rumpf-Ausdrücken ein

Symbol verwendet, das im **let** gebunden wurde, wird bei der Auswertung der mit **let** definierte Wert verwendet:

```
>> (let [a 2
         b (* a a)
         c (* b b)]
     (+ a b c))

==> (+ 2 4 16) ==> 22
```

Die durch **let** erzeugte Umgebung hat lexikalischen Gültigkeitsbereich. In geschachtelten **let**-Ausdrücken sind die Bindungen der äußeren Ausdrücke sichtbar, können aber überdeckt werden:

```
>> (let [a 2]
     (let [a (* a a)
           b (* a a)]
       (println "a innen: " a)
       (println "b innen: " b))
     (println "a aussen: " a))
 ==> Ausgabe: "a innen: 4", "b innen: 16", "a aussen: 2"
```

Schleifen mit loop und recur

Für die wiederholte Auswertung von Ausdrücken stellt Clojure die **loop**-Sonderform bereit. **loop** hat eine ähnliche Syntax wie **let**:

```
(loop [<symbol> <bindungs-ausdruck> ...] <rumpf-ausdruck> ...)
```

loop erzeugt wie **let** eine Umgebung und wertet die Rumpfausdrücke in dieser Umgebung aus.

Als Sonderfall ist innerhalb von **loop** ein **recur**-Aufruf erlaubt, dem so viele Ausdrücke übergeben werden, wie Bindungen im **loop**-Aufruf vorhanden sind. **recur** erzeugt eine neue Umgebung, in der die Symbole an die in **recur** übergebenen Werte gebunden sind, und wertet die Rumpfausdrücke in dieser Umgebung erneut aus. Eine wichtige Einschränkung ist, dass ein **recur**-Aufruf nur als letzter auszuwertender Ausdruck der Umgebung verwendet werden darf (an der „tail position“, erinnert an Endrekursion).

Hier eine zulässige Nutzung von **recur**:

```
>> (loop [i 0]
     (println i)
     (if (< i 5)
       (recur (+ i 1))
       (println :fertig)))
==> Ausgabe: 0 1 2 3 4 5 :fertig
```

Die folgende Schleife ist nicht korrekt und wird von Clojure mit einer **UnsupportedOperationException** quittiert. Die Ursache ist der **println**-Ausdruck, der als letzter Ausdruck der **loop** auszuwerten ist.

```
>> (loop [i 0]
     (println i)
     (if (< i 5)
       (recur (+ i 1)))
     (println "Aufruf in \"loop\" nach recur"))
```

Funktionen sind auch nur Werte

Funktionen sind in Clojure Werte. Sie können einer Funktion als Parameter übergeben werden und als Wert von einer Funktion zurückgegeben werden. Dadurch können Funktionen geschrieben werden, die bestehende Funktionen zu einer neuen Funktion kombinieren.

Ein Beispiel ist **compose**. Diese Funktion erhält zwei Funktionen **f** und **g** als Parameter und erzeugt eine Funktion, die **f** auf das Ergebnis von **g** anwendet:

```
>> (defn compose [f g]
     (fn [x] (f (g x))))
>> (defn quadrat [x] (* x x))
>> (defn eins-durch [x] (/ 1 x))
>> ((compose quadrat eins-durch) 4) ==> 1/16
```

Funktionen, die Funktionen als Werte übernehmen bzw. als Ergebnis liefern, werden oft als „Funktionen höherer Ordnung“ bzw. „Higher Order Functions“ bezeichnet.

Namespaces und Libs

Gute Namen für etwas zu finden ist schwierig, und oft wird ein Name schon an anderer Stelle verwendet. Clojures Lösung für dieses Problem sind „Namespaces“. Mit

```
(ns <Namespace-Name> <Externe Referenzen>*)
```

öffnen Sie einen Namensraum, einen sogenannten Namespace. Alle folgenden Definitionen sind Elemente dieses Namensraums. Der aktuelle Namensraum ist stets über das Symbol ***ns*** zugänglich.

Namensräume verhalten sich ähnlich wie Java-Packages. Ein Name, der innerhalb eines Namensraums definiert wurde, kann ohne Nennung des Namensraums verwendet werden. Namen aus anderen Namensräumen müssen entweder mit dem vollqualifizierten Namen **<Namespace-Name>/<Lokaler Name>** referenziert werden oder aber das Element muss als externe Referenz importiert worden sein. Für den Import von Elementen aus anderen Namensräumen und Java-Packages existieren in Clojure verschiedene Formen.

Mit **(ns <mein-namensraum (:use [<externer-namensraum>])** werden alle öffentlichen Definitionen aus **<externer-namensraum>** in **<mein-namensraum>** über den unqualifizierten Namen zugänglich gemacht. Die importierten Namen können optional durch Angabe von **:exclude (name …)** bzw. **:only (name…)** eingeschränkt werden. Bei Namenskollisionen im importierenden Namensraum wird eine Exception geworfen.

Damit Definitionen aus einem Namensraum mit **use** importiert werden können, müssen sie in einer „Lib“ enthalten sein. Eine Lib ist eine Ressource im Klassenpfad der virtuellen Maschine, deren Pfad nach folgenden Regeln abgeleitet sein muss:

- Punkte „.“ im Namensraum werden zu „/“ im Pfad.
- Minus „-“ im Namensraum werden zu „_“ im Pfad.
- Der Pfad muss mit „.clj“ enden.

Zur Demonstration noch ein kurzes Beispiel: Der Namensraum **com.innoq.a-namespace** muss in der Ressource **/com/innoq/a_namespace.clj** definiert sein:

```
;; Die folgenden Definitionen als Ressource
;; /com/innoq/a_namespace.clj im Klassenpfad
(ns com.innoq.a-namespace)
(defn mal2  [x] (* 2 x))
(defn kubik [x] (* x x x))
```

Der Import der Funktion **kubik** in einem anderen Namensraum sieht dann so aus:

```
>> (ns com.innoq.b
     (:use [com.innoq.a-namespace :only (kubik)]))
>> (kubik 3) ==> 27
```

```
>> (mal2 3) ==> Exception. Unable to resolve symbol: mal2 in
this context          ;; Es wurde nur „kubik“ importiert.
```

Ausblick

Im ersten Teil haben wir die grundlegende Syntax von Clojure vorgestellt. Im folgenden Teil werden Clojures Datenstrukturen vorgestellt und die Java-Integration erläutert.

Links

[clojure] Startseite des Clojure-Projekts, R. Hickey, 2008-2010,
`http://clojure.org`
[clojureAPI] Dokumentation des Clojure-API, R. Hickey, 2007,
`http://richhickey.github.com/clojure`
[clojure1.1.0] Clojure-Quellen und jar-Archiv,
`http://clojure.googlecode.com/files/clojure-1.1.0.zip`
[counterclockwise] Eclipse-Plug-In für Clojure,
`http://code.google.com/p/counterclockwise/`

Burkhard Neppert ist Senior Consultant bei der innoQ Deutschland GmbH.
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.