inkl. CD

Android User Interfaces: Grundlagen und Tipps »64

S&S

JAVA Mag

Deutschland €8,50 Österreich €9,80 Schweiz sFr 16,80 2.2011

Javamagazin™

Java • Architekturen • Web • Agile

www.javamagazin.de

CD-INHALT

jax TV

ADAM BIEN
60 Minuten mit
Java EE

Video von der JAX 2010
in voller Länge

HIGHLIGHTS

Java ist auch
eine Insel als
OpenBook

Android SDK
Clojure 1.2

WEITERE INHALTE

EasyMock 3.0
jCouchDB
ScalaTest
ScalaCheck u.v.

Alle CD-Infos ab Seite 3

Clean Code
Design Pattern
Teil 3 »45

RHQ
Auf die Finger
geschaut »82

jax 2011
Alle Infos im Heft »35

Clojure

Multithreading auf der JVM

Clojure: Wozu wieder eine neue
Programmiersprache? »20-34

Interview mit Clojure-
Gründer Rich Hickey »16

CouchDB
Bequemer Zugriff aus Java »75

TDD mit Scala
Mit guten Vorsätzen ins neue Jahr »97

Scrum But(t)
Wie viel Anpassung ist gut für Scrum? »110

4 194586 708507 02

Datenträger enthält
Info- und
Lehrprogramme
gemäß § 14 JuSchG

Lösungsansätze für typische Fragen im Programmieralltag

Auf die Finger geschaut

Wenn man als Java-Entwickler nur lange genug in Ruby programmiert, sammeln sich automatisch so viele runde Klammern an, dass man sich fragen muss, wie man die wieder los wird. Eine geeignete Möglichkeit ist, eine Weile lang in einer Lisp-ähnlichen Programmiersprache wie Clojure zu programmieren. In diesem Artikel möchte ich ein paar Lösungsansätze für gängige Fragestellungen im Programmieralltag präsentieren. Vielleicht bringt der Artikel den einen oder anderen Lisp-Guru zum Weinen oder Lächeln. Immerhin entspringt er der Tastatur eines C++-/Java-/Ruby-Fuzzis.

von Phillip Ghadir

Ich möchte im Wesentlichen auf existierende Problemstellungen zurückgreifen, die uns in verschiedenen Projekten bereits begegnet sind. In seinem Clojure-Einführungsartikel (siehe S. 25) hat uns Burkhard Neppert erläutert, wie wir Listen, Vektoren und Maps verwenden können.

Aus einem Vektor eine Hash-Map

Nehmen wir an, dass wir eine Sequenz Nutzdaten verarbeiten wollen, die in Vektoren abgelegt sind, beispielsweise:

```
(def revenues
   (list
    ["NRW" 2010 01 "Ghadir, Phillip" 1500]
    ["NRW" 2010 01 "Tilkov, Stefan" 1500]
    ["NRW" 2010 01 "Neppert, Burkhard" 1500]
    ["NRW" 2010 02 "Ghadir, Phillip" 700]
    ["NRW" 2010 02 "Tilkov, Stefan" 2300]
    ["NRW" 2010 02 "Neppert, Burkhard" 1000]))
```

Um im Verlauf unseres Programms nicht ständig darauf angewiesen zu sein, den korrekten Index zu kennen, haben wir verschiedene Möglichkeiten, den Zugriff zu kapseln. Die geradlinige Variante wäre, Konstanten zu definieren, die den Index des jeweiligen Werts kapseln, z.B. in folgender Form:

```
(def REGION 0)
(def YEAR 1)
```

Der Nachteil ist, dass man sich im Folgenden auf eine Zugriffsmethode verlassen wird, die mit diesen Schlüsseln umgehen können muss. Sollte sich im zuliefernden System die Datenstruktur ändern, kann das unerwünscht viele Änderungen notwendig machen.

Dieser Gefahr beugen wir vor, indem wir die Daten direkt in eine wohl strukturierte Form transformieren, z. B. in eine Hash-Map. Die Methode *bind* zeigt, wie dies für einen jeden Vektor aus der obigen Liste der *revenues* geschehen kann:

```
(defn bind [[region year month employee revenue]]
  {
   :region region,
   :year year,
   :month (str year "-" month),
   :employee employee,
   :revenue revenue
   })
```

Die Funktion *bind* sollten wir uns genauer anschauen: Zuerst fällt auf, dass die Parameterliste von *bind* irgendetwas bekommt, was wie ein Vektor aussieht, d. h. Werte stehen in eckigen Klammern geschrieben. Immer wenn wir so etwas in einer Parameterliste sehen, nennt man das Destrukturierung (Destructuring). Anstatt einen Parameter zu deklarieren, der den übergebenen Wert (also den ganzen Vektor) enthält, können wir definieren, dass die Elemente des Vektors auf lokale Variablen abgebildet werden. **Abbildung 1** veranschaulicht die Destrukturierung. Allein in der Signatur von *bind* ist nun festgelegt, in welcher Reihenfolge wir die Daten erwarten. Destrukturierung ist nicht nur auf Vektoren bekannter Länge beschränkt. Unter [1] gibt es eine Erläuterung, wie Destrukturierung mit beliebig langen Vektoren oder Hash-Maps funktioniert.

Unsere *bind*-Funktion liefert eine Hash-Map mit den Werten zurück und transformiert den Wert für Monat, sodass dieser allein ausreicht, um Daten aus mehreren Jahren anschließend zu gruppieren oder selektieren. Wie auf S. 25 ff. beschrieben, werden hier von uns festgelegte Schlüsselwörter (*:region*, *:employee* etc.) als Schlüssel verwendet. Die Hash-Map wird einfach zurückgegeben. Sollte sich also irgendwann die Struktur von *revenues* ändern, genügt es, die Funktion *bind* anzupassen. Das Transformieren der gesamten Liste erfolgt dann z. B. per *(map bind revenues)*. Das Ergebnis entspricht dann dem, was wir im Folgenden verarbeiten können:

```
({:region "NRW", :year 2010, :month "2010-1",
  :employee "Ghadir, Phillip", :revenue 1500}
 {:region "NRW", :year 2010, :month "2010-1",
  :employee "Tilkov, Stefan", :revenue 1500}
 {:region "NRW", :year 2010, :month "2010-1",
  :employee "Neppert, Burkhard", :revenue 1500}
 {:region "NRW", :year 2010, :month "2010-2",
  :employee "Ghadir, Phillip", :revenue 700}
 {:region "NRW", :year 2010, :month "2010-2",
  :employee "Tilkov, Stefan", :revenue 2300}
 {:region "NRW", :year 2010, :month "2010-2",
  :employee "Neppert, Burkhard", :revenue 1000})
```

Level 2 – Konstruktion und Verwendung von hierarchischen Datenstrukturen

Wollen wir diese Daten nun verarbeiten, gibt es die Möglichkeit, die *revenues* einfach aufzuaddieren. Dazu definieren wir eine kleine Hilfsfunktion, um nicht zweimal über die Menge aller *revenues* gehen zu müssen: *(defn sum-revenue [x y] (+ x (:revenue y)))*.

Im Anschluss können wir direkt *reduce* verwenden, so wie es uns der Grundlagenartikel (siehe S. 25) erklärt hat. Dazu definieren wir eine Hilfsfunktion *sum-revenues*, die mit *reduce* über die Liste von Hash-Maps geht und einen definierten Wert (hier *revenue*) aufaddiert:

```
(defn sum-revenues [h]
  (reduce sum-revenue 0 h))
```

Der Aufruf erfolgt dann per *(sum-revenues (map bind revenues))*. Damit erhalten wir auf der Menge aller *revenues* die Summe. Möchten wir die Menge einschränken, bietet sich dafür die Funktion *filter* an. Dazu merken wir uns eben die transformierten *revenues*: *(def revs (map bind revenues))*. Jetzt können wir z. B. nur die Einnahmen von „Ghadir, Phillip" nehmen und aufsummieren:

```
(sum-revenues
  (filter
    #(= (:employee %) "Ghadir, Phillip")
    revs))
```

Die Funktion *filter* erhält zwei Argumente. Das erste ist eine Funktion, die für ein Argument einen Wahrheitswert (*true* oder *false*) liefert. Diese Funktion nennt man Prädikat. Die Funktion *filter* ruft das Prädikat auf jedem Element des zweiten Arguments auf. In die Ergebnismenge werden nur die Elemente aufgenommen, für die das Prädikat erfüllt ist (also *true* liefert). Wenn wir ein Prädikat öfter benötigen, können wir diese in Clojure bequem zusammenbauen:

```
(defn matches [x y] (fn [z] (= (z x) y)))
(def isPhillip (matches :employee "Ghadir, Phillip"))
```

Mit einem so benamten Prädikat wird der *filter*-Ausdruck gleich lesbarer und das Aufsummieren der Einnahmen von Phillip folgendermaßen geschrieben: *(sum-revenues (filter isPhillip revs))*.

Einfache Aggregation

Möchten wir die Einnahmen nach Mitarbeitern gruppiert berechnen, können wir auf Teile unserer bisherigen Implementierung zurückgreifen. Dank der höherwertigen Funktionen stehen uns in Clojure einfache Mittel zur Verfügung, die wir nur neu zu kombinieren brauchen. Um zum Beispiel die Einnahmen nach Mitarbeitern zu gruppieren, können wir z. B. Folgendes schreiben:

```
(reduce
 (fn [a {e :employee, r :revenue }]
    (merge-with + a {e r}))
 {} revs)
```

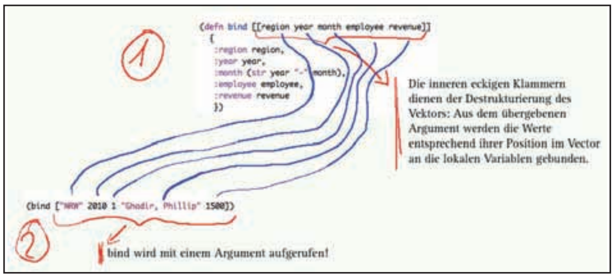
Die Funktion *reduce* erwartet drei Parameter:

- eine Funktion, mit der aus zwei Werten einer gemacht wird
- ein initialer Wert für den Akkumulator
- die Menge, die reduziert werden soll

Als Erstes erhält *reduce* eine Funktion, die zwei Argumente erwartet. Das Erste ist der Akkumulator, mit dem die bisherigen Ergebnisse weitergereicht werden. Das zweite Argument benennen wir gar nicht erst, sondern zerlegen es mittels Destrukturierung. Wir erwarten hier eine Map, deren Werte für die Schlüssel *:employee* und *:revenue* direkt in lokalen Variablen *e* und *r* gespeichert werden. Mit der Zeile *(merge-with + a {e r})* rufen wir nun *merge* mit der Akkumulator-Hash-Map *a* und einer Hash-Map mit einem Schlüssel *e* und dem zugehörigen Wert *r* auf. Sollte in *a* bereits ein Wert für den Schlüssel *e* enthalten sein, speichere statt *r* das Ergebnis von *(+ (a e) r)* – zu einfach!

Das zweite Argument von *reduce* ist dann eine leere Hash-Map als initialer Akkumulator. Das dritte Argument ist eine Menge von *revenue-hash*-Strukturen – hier die ungefilterte Menge *revs*. Wenn wir also die obige Funktion abstrahieren wollen, können wir eine Funktion schreiben, mit der wir hinterher einfacher zu verwendende Funktionen definieren können:

```
(defn aggregate-revenues-by [k1]
  (fn [revs] (reduce
     (fn [a {e k1, r :revenue }]
```



```
(defn bind [[region year month employee revenue]]
  {
   :region region,
   :year year,
   :month (str year "-" month),
   :employee employee,
   :revenue revenue
   })
```

Die inneren eckigen Klammern dienen der Destrukturierung des Vektors: Aus dem übergebenen Argument werden die Werte entsprechend ihrer Position im Vector an die lokalen Variablen gebunden.

```
(bind ["NRW" 2010 1 "Ghadir, Phillip" 1500])
```

bind wird mit einem Argument aufgerufen!

Abb. 1: Automatische Destrukturierung eines Vektors

```
  (merge-with + a {e r}))
    {} revs)))
```

Die Funktion *aggregate-revenues-by* erwartet als Argument einen gewünschten Schlüssel, nach dem aggregiert werden soll, und liefert eine Funktion, die wie zuvor beschrieben, die Einnahmen nach dem gewünschten Schlüssel gruppiert und aufsummiert haben. Die Funktion, die von *aggregate-revenues-by* zurückgeliefert wird, verwendet intern eine Reduktionsfunktion, bei der das zweite Argument wie zuvor destrukturiert wird. Diesmal werden allerdings die Werte von *:revenue* und von dem zuvor festgelegten Schlüssel ausgelesen. Zu den einschlägigen Schlüsselwerten können wir uns damit direkt ein paar neue Aggegrationsfunktionen bauen:

```
(def aggregate-revenues-by-month
    (aggregate-revenues-by :month))
(def aggregate-revenues-by-employee
    (aggregate-revenues-by :employee))
```

Hier ist zu beachten, dass wir mit *aggregate-revenues-by-month* und *...-by-employee* Funktionen definiert haben, die beide nur ein Argument erwarten: die Menge der Hash-Maps mit Einnahmen. Die Ausgabe aller Einnahmen aus 2010 gruppiert nach Mitarbeitern sieht damit folgendermaßen aus:

```
(aggregate-revenues-by-employee (filter (matches :year 2010) revs))
```

Und das Ergebnis entspricht nun unseren Erwartungen:

```
{ "Neppert, Burkhard" 2500,
  "Tilkov, Stefan" 3800,
  "Ghadir, Phillip" 2200 }
```

Möchten wir nun gleich mehrere Prädikate innerhalb eines Filters anwenden, um eine Menge mit einem Durchlauf zu reduzieren, können wir das sehr einfach realisieren, in dem wir eine zusammengesetzte Prädikatsfunktion implementieren.

Wir erinnern uns: Ein Prädikat liefert für ein Argument einen Wahrheitswert. Wir können also nicht einfach *(and (matches :employee "Ghadir, Phillip") (matches :year 2010))* schreiben. Denn in diesem Fall würde nur überprüft, ob die beiden Argumente von *and* wahr sind. Wir haben hier aber Funktionen hinein gegeben. Diese sind per Definition ungleich *nil* und werden damit als *true* angenommen. Wir müssen deshalb innerhalb des zusammengesetzten Prädikats die einfachen Funktionen aufrufen. In dem folgenden Filterausdruck wird dies inline gemacht:

```
(filter
 #(and
   ((matches :employee "Ghadir, Phillip") %)
   ((matches :month "2010-2") %))
 revs)
```

Die Argumente von *and* sind nun nicht die durch *matches* erzeugten Funktionen, sondern das Ergebnis deren Aufrufs mit dem Parameter (siehe hierzu auch den Einführungsartikel, S. 25).

Fazit und Ausblick

Selbst ein Laie wie ich bekommt mit Clojure schon den einen oder anderen sinnvollen Algorithmus implementiert. In diesem Artikel haben wir gesehen, mit wie wenig Quellcode Daten aggregiert werden können. Ohne auf die bereits existierenden Funktionen zurückgegriffen zu haben, waren wir dennoch in der Lage, Listen von Daten zu transformieren und zu verwenden. Wem die Verwendung von Listen, Vektoren und Hash-Maps zu (typ-)unsicher erscheint, kann auf strukturiertere Datenstrukturen zurückgreifen.

Aus der imperativen Welt sind wir gewohnt, selbst zu schreiben, wie über eine Datenstruktur iteriert wird. Mit imperativen Sprachkonstrukten wären viele der hier vorgestellten Konstruktionen aus Laufzeitsicht katastrophal. Anscheinend durchlaufen wir an manchen Stellen mehrfach dieselbe Datenmenge, nur um andere Facetten auszuwerten. Glücklicherweise rettet uns Clojure hier durch die Verwendung der Abstraktionssequenz (in Clojure *seq* genannt). Damit ist es möglich, auch ineinander geschachtelte Funktionsaufrufe zu implementieren, bei denen bereits Funktionen auf die ersten Elemente einer Sequenz aufgerufen werden, bevor der innerste Aufruf (z. B. *map*) vollständig ausgewertet wurde. Aus Architekturgesichtspunkten entspricht die Aneinanderreihung solcher Funktionen einer Pipeline-Architektur [2].

Phillip Ghadir ist CTO und Principal Consultant bei innoQ. Er sucht nach Wegen, die Softwareentwicklung effizienter zu gestalten und hat als Softwarearchitekt die Konzeption und Konstruktion verschiedener unternehmenskritischer, verteilter Systeme begleitet.

Links & Literatur

[1] http://clojure.org/special_forms#Special%20Forms--%28let%20[bindings*%20]%20exprs*%29

[2] Buschmann, Frank et al: „Pattern orientierte Softwarearchitektur", Addison-Wesley 2000

[3] http://clojure.org

[4] Neppert, Burkhart; Tilkov, Stefan: „Clojure: Funktional, parallel, genial – Teil 1: Überblick, JavaSPEKTRUM 2.10

[5] Neppert, Burkhart; Tilkov, Stefan: „Einführung in Clojure – Teil 2", JavaSPEKTRUM 3.10