Totally Lazy

Java – Funktional ohne Zauberei

Phillip Ghadir

In dieser Ausgabe stelle ich die verschiedenen Möglichkeiten vor, mit der quelloffenen Bibliothek Totally Lazy in Java funktional zu programmieren. Mit Hilfe bekannter Higher-Oder-Functions erlaubt sie, Collections elegant sequenziell oder auch nebenläufig zu verarbeiten. Dabei stellt Totally Lazy sicher, dass die Verarbeitung der Sequenzen lazy erfolgt. Wie Totally Lazy hilft, Teile unseres Quellcodes wartungsfreundlicher zu gestalten, ist Gegenstand dieses Artikels.

Funktional Programmieren in Java

Derzeit erhält das funktionale Programmieren [Wamp11] sehr viel Beachtung – nicht nur in der Community, sondern auch in dieser Kolumne. Während sich verschiedene Beiträge eher um funktionale Aspekte der JVM-Sprachen wie Clojure oder Scala drehen, stellt dieser Artikel die Möglichkeiten der Bibliothek Totally Lazy [total] vor, die vollständig ohne eigene Compiler oder Laufzeit-Zaubereien zurecht kommt.

Totally Lazy

Das Grundprinzip der überschaubaren Bibliothek ist simpel. Mit Hilfe von Schnittstellen bildet das Framework die Abstraktionen, die nötig sind, um gängige Higher-Order-Functions wie zum Beispiel `map` und `reduce` zu unterstützen.

Totally Lazy definiert diese Abstraktionen mit Generics, sodass beim Hintereinanderschalten von Funktionen die Typsicherheit gewährleistet bleibt und wir weiterhin in der Entwicklungsumgebung unserer Wahl mit *<Strg>+<SPACE>* programmieren können.

Funktionsparameter für die Higher-Order-Functions sind zu einem großen Teil bereits vordefiniert, können aber genauso gut selbst implementiert werden. Dazu muss man nur die entsprechende Schnittstelle realisieren, die von der Higher-Order-Function als Parametertyp erwartet wird.

Das Kernkonzept von Totally Lazy ist die verzögerte Auswertung von Sequenzen, ähnlich wie dies auch in Clojure geschieht. Das bedeutet, solange keine Werte aus einer Sequenz konkret gebraucht werden, wird auch nicht über die Sequenz iteriert.

Dazu verwendet Totally Lazy eine eigene Datenstruktur `Sequence`, die selbst `java.lang.Iterable` realisiert. Higher-Order-Functions liefern stets `Sequence`-Instanzen zurück, wenn als Ergebnis eine Menge von Werten erwartet wird.

Ulkiger funktionaler Programmcode

Wir Java-Entwickler sind es häufig gewohnt, über Listen zu iterieren.

```
for ( Element e : meineListe ) {
  // mach etwas Intelligentes mit e
}
```

Dies würde man mit Totally Lazy schreiben können:

```
meineListe_ist_jetzt_eine_Sequence.each(
  new IntelligenteVerarbeitungVonE() );
```

Totally Lazy – Kurzüberblick

Die Konvertierung von Java-Collections zu Totally-Lazy-Sequenzen und zurück:

```
java.util.List<String> liste = Arrays.toList( "A", "B", "C", "D" );

Sequence<String> seq1 = sequence( "A", "B", "C", "D" );
Sequence<String> seq2 = sequence( liste );

java.util.List<String> listeNeu = seq1.toList();
```

Funktionen, die Sequenzen zurück liefern, werden *lazy* ausgewertet:

```
// liefert eine noch nicht ausgewertete Sequenz mit 2,4:
sequence(1, 2, 3, 4).filter(even);
// liefert eine noch nicht ausgewertete Sequenz mit "1", "2":
sequence(1, 2).map(toString);
// liefert eine Sequenz, die bei Auswertung die Arbeit
// an Hintergrund-Threads delegiert:
sequence(1, 2).mapConcurrently(toString);
// liefert eine noch nicht ausgewertete Sequenz mit 1,2:
sequence(1, 2, 3).take(2);
// liefert eine noch nicht ausgewertete Sequenz mit 3:
sequence(1, 2, 3).drop(2);
// liefert eine noch nicht ausgewertete Sequenz mit 2,3:
sequence(1, 2, 3).tail();
```

Funktionen, die einen anderen Rückgabetyp haben, werden *eager* ausgewertet:

```
// liefert das erste Element: 1:
sequence(1, 2, 3).head();
// addiert alle Zahlen der Sequenz auf und liefert 6:
sequence(1, 2, 3).reduce(sum);
sequence(1, 3, 5).find(even); // liefert none()
sequence(1, 2, 3).contains(2); // liefert true
sequence(1, 2, 3).exists(even); // liefert true
sequence(1, 2, 3).forAll(odd); // liefert false;
// addiert alle Zahlen zur 0 und liefert 6:
sequence(1, 2, 3).foldLeft(0, add);
sequence(1, 2, 3).toString(); // liefert "1,2,3"
sequence(1, 2, 3).toString(":"); // liefert "1:2:3"
```

Mit Hilfe sogenannter Generatoren können Sequenzen erzeugt werden:

```
// liefert eine noch nicht ausgewertete Sequenz mit1,2,3,4:
range(1, 4);
// liefert eine noch nicht ausgewertete unendliche Sequenz von
// "car":
repeat("car");
// liefert eine noch nicht ausgewertete Sequenz mit
// 1,2,3 ... bis unendlich:
iterate(increment, 1);
// liefert eine noch nicht ausgewertete unendliche Sequenz mit
// 1,2,3,4,1,2,3,4,1,2,3,4,...:
range(1, 4).cycle();
// liefert eine noch nicht ausgewertete Sequenz der Primzahlen:
primes();
// liefert eine noch nicht ausgewertete Sequenz der Fibonacci-
// Zahlen:
fibonacci();
// liefert eine noch nicht ausgewertete Sequenz mit 3^x
// (1,3,9,27 ...):
powersOf(3);
```

Dabei würden wir den Block aus der oberen for-Schleife nun in eine eigene Klasse namens **IntelligenteVerarbeitungVonE** in die Methode **call(Element e)** auslagern. Alternativ könnten wir auch (da wir noch keine Lambdas zur Verfügung haben), direkt eine anonyme innere Klasse instanziieren:

```
meineListe_ist_jetzt_eine_Sequence.each(
  new Callable1<Element, Object>() {
    @Override
    public Object call(Element e) {
      return null; // TODO mach etwas Intelligentes mit e
    }
  }
);
```

Für die einfachen Beispiele können wir also festhalten, dass sich der Quelltext nicht unbedingt verbessert. Der Artikel wäre also hiermit am Ende, hätte das Ganze nicht noch etwas Gutes.

Komplexität zerlegbar machen

Auch wenn die dritte Variante insgesamt fünf Zeilen Programmcode brauchte (und drei Zeilen mit schließenden Klammern), um etwas zu schreiben, was man mit foreach auch in drei Zeilen hätte schreiben können, kann Totally Lazy zur Lesbarkeit von Programmfragmenten beitragen. Beispielsweise kennen Sie sicherlich in Ihren Programmen ähnliche Codefragmente:

```
List ergebnisListe = ...;
for ( Kunde k : kundenliste ) {
  if ( k.istVIPKunde() ) {
    //  ...
    ergebnisListe.add( ... );
  }
}
return ergebnisListe;
```

Dies kann man mit Totally Lazy einfacher ausdrücken. Zur Erhöhung der Lesbarkeit sind die anonymen inneren Klassen nicht inline, sondern vorab instanziiert und den Variablen **istVIPKunde** sowie **intelligenteVIPVerarbeitung** zugewiesen. Dann liest sich obiges Fragment so:

```
return kundenlist e.filter( istVIPKunde )
.map(intelligenteVIPVerabeitung );
```

Auch wenn die Verwendung einer solchen Fluent-API aussieht, als würde mehrfach über die Liste iteriert werden, ist dem nicht so. Über die Elemente der Kundenliste wird einmal iteriert. Für jedes Element, das dem Prädikat **istVIPKunde** genügt, ruft **map** das Callable-Objekt **intelligenteVIPVerarbeitung** auf und sammelt das Ergebnis in der Ergebnissequenz.

Vorteilhafte Struktur

Da Totally Lazy intern auf **java.lang.Iterable** und **java.util.Iterator** setzt, kann die Ergebnisauswertung verzögert werden. Dies erlaubt gleichermaßen das Definieren unendlicher Folgen wie die partielle Auswertung von langen Sequenzen mit beliebig vielen Elementen. Zudem können die Funktionen von Totally Lazy direkt auf den **java.util.Collection**s angewendet werden. Statt

```
sequence.<funktion>( ... )
```

schreibt man dann mit den richtigen statischen Imports

```
<funktion>( JavaCollection, ... ).
```

Die Funktionen von Totally Lazy kennt man aus verschiedenen anderen Sprachen – zum Beispiel aus ML, Haskell oder Lisp. Dadurch ist das Lesen von Quellcode ohne größere Überraschungen möglich. Einen Überblick über einige der Funktionen bietet der Kasten „Totally Lazy – Kurzüberblick“.

Es wird weder ein spezielles Instrumentieren von Klassen benötigt, noch propagiert Totally Lazy Informationen über ThreadLocal. Für die Wartung genügt also ein grundlegendes Verständnis der Higher-Order-Functions und der „Funktionen“, die als Parameter übergeben werden.

In der Java-Welt zuhause

Die Bibliothek zeigt, wie weit funktionale Konzepte in der Implementierung verwendet werden können. Meine bevorzugte Entwicklungsumgebung stellt den Aufruf auch noch kompakt dar, sodass der Code dank Codevervollständigung nicht nur schnell geschrieben ist, sondern auch noch leicht gelesen werden kann. Abbildung 1 zeigt anhand des letzten Codefragments aus „Ulkiger funktionaler Programmcode“ (s. diese Seite oben links), wie die Entwicklungsumgebung die Instanziierung und Definition einer anonymen inneren Klasse auf das Wesentliche reduziert.

Abb. 1: Code-Folding in IntelliJ Idea

Das ist nicht mehr weit weg von dem, was man in anderen Programmiersprachen lesen würde. Ein Klick auf das „+“ expandiert den Text zur vollständigen Form. Darüber hinaus kann man das Aufbereiten der Ergebnisliste den Higher-Order-Functions überlassen. Java-Methoden lassen sich dann besser lesen, wenn der Teil des Quelltextes entfällt, der sonst für den Listenaufbau benötigt würde.

Der Klassiker mit Totally Lazy implementiert

Wenn wir beispielsweise die Umsätze unserer Top-10-Kunden – sortiert nach Kundennamen – ausgeben sollten, sähe die Implementierung unserer Logik recht einfach aus:

```
return map( kundenliste, toGenericMap() )
.sortBy( comparatorOnKeys( "umsatz" ) )
        .take( 10 )
        .sortBy( comparatorOnKeys("name", "kundenNr") );
```

Klar, wer eine relationale Datenbank in Reichweite hat, könnte genauso gut diese das Filtern übernehmen lassen.

Weitere Funktionen

Häufig wollen wir auch die restlichen Elemente einer Sequenz sinnvoll weiterverarbeiten. Mit der Funktion **partition** lassen sich Sequenzen einfach in zwei Gruppen unterteilen: die erste Gruppe mit all den Elementen, für die das angegebene Prädikat true liefert. Die zweite Gruppe beinhaltet den Rest.

```
Pair<Sequence<Kunde>, Sequence<Kunde>> partitionen =
        kundenliste.partition( istVIPKunde() );
```

Hier kann die individuelle Verarbeitung erfolgen. Für die VIP-Kunden könnte eine Einzelsatz-Behandlung anstehen:

```
partitionen.first().each( spezielleVIPBehandlung() );
```

Während das Gros der Kunden über den großen Retail-Kamm geschert wird:

```
abZurSpamFarm( partitionen.second() );
```

Wem die funktionale Schreibweise zu aufdringlich ist und zu wenig nach Java-Slang aussieht, kann ohne Weiteres an Methoden delegieren, die selbst wiederum gewohntes Java verwenden und beispielsweise mit foreach über die Collections iterieren.

Sie wollen Elemente durchnummerieren? – Mit Totally Lazy ist das kein Problem. Die Funktion **zip** erlaubt es, zwei Sequenzen zusammenzufügen. Daraus resultiert eine Sequenz, die so lang ist wie die kürzere der beiden Sequenzen und als Elemente jeweils ein Wert-Paar enthält. Der erste Wert stammt aus der ersten Sequenz, der zweite aus der zweiten. Mit

```
zip( iterate( increment(), 1), kundenliste );
```

erhalten wir also eine Sequenz von Paaren:

```
[1, Kunde@Instanz1], [2, Kunde@Instanz2], [3, Kunde@Instanz3]
```

die natürlich erst ausgewertet wird, wenn die Werte benötigt werden.

Paralleles Verarbeiten von Sequenzen

Totally Lazy bietet Funktionen an, deren Namen mit **Concurrently** enden, um die Verarbeitung von Sequenzen nebenläufig auszuführen. Dies ist sinnvoll, wenn die Verarbeitung eines Elements potenziell lang dauert, aber aufgrund der Rechnerarchitektur auch parallelisiert werden kann.

Die auf **Concurrently** endenden Methoden verwenden intern einen **java.util.concurrent.Executor**, um die Funktionsparameter der Higher-Order-Functions nebenläufig aufzurufen. Dies kann bei lang laufenden Funktionen die Durchlaufzeit erheblich verkürzen. Um die Auswertung der Sequence zu erzwingen, können wir unter anderem die Funktion **forAll** verwenden.

Der Vergleich im Kleinen – auf einer Maschine mit vier Kernen – lässt sich sehen. Mit folgendem (Java7)-Code wird zweimal die Sequenz **1,2,3** mit der Funktion **veryLongRunningFunction** verarbeitet. Dabei wird die Zeit innerhalb des **try**-Blocks mit einer eigens geschriebenen Hilfsklasse **StopWatch** gemessen. Ist der **try**-Block durchlaufen, gibt die **StopWatch** automatisch die Zeit aus:

```
try (StopWatch watch = new StopWatch("Sequentially", System.out)) {
  sequence(1,2,3).map(veryLongRunnningFunction)
        .forAll(theYesPredicate);
}

try (StopWatch watch = new StopWatch("Concurrently", System.out)) {
  sequence(1,2,3).mapConcurrently(veryLongRunnningFunction)
        .forAll(theYesPredicate);
}
```

Der Quelltext in beiden Blöcken unterscheidet sich nur in dem Namen der **StopWatch** sowie in dem Aufruf der Higher-Order-Function. Im ersten Block wird das sequenzielle **map** aufgerufen, im zweiten das nebenläufige **mapConcurrently**. Die Methode **forAll** erzwingt die Auswertung der Sequence. Die Ausführung innerhalb eines kleinen Tests ergibt:

```
StopWatch 'Sequentially' took 3003395000 ns
StopWatch 'Concurrently' took 1004120000 ns
```

Die nutzlose Wartezeit in der **veryLongRunningFunction** wird durch die Nebenläufigkeit also besser genutzt.

Eigene Funktionen definieren

Der Kasten „Totally Lazy – Kurzüberblick“ bietet eine kleine Übersicht über die gängigen Funktionen/Methoden, die Totally Lazy bietet, um Sequenzen zu verarbeiten. Wem das nicht reicht, der kann mit denselben Mitteln eigene Funktionen definieren, ohne dabei auf Drittbibliotheken angewiesen zu sein.

Totally Lazy auf Steroiden

Wer bereit ist, zur Laufzeit seinen Bytecode instrumentieren zu lassen oder einen Zwischenschritt beim Build einzufügen, der die Bytecode-Manipulation übernimmt, kann mit Hilfe der zusätzlichen Bibliothek Enumerable [enumerable] Lambdas verwenden. Im Kontext von Enumerable sind Lambdas anonyme Funktionen mit nur einem Ausdruck. Darin unterscheiden sich die Lambdas von Enumerable zum Beispiel von Blöcken in Ruby.

Wenn wir Laufzeit-Instrumentierung von Enumerable einbinden, können wir die JVM mit der zusätzlichen **javaagent**-Direktive starten:

```
java -javaagent:enumerable-java-<version>.jar …
```

Dann können wir in unserem Java-Code auch Abkürzungen schreiben wie:

```
Fn1 square = λ( n, n * n);
```

anstatt dies ausschreiben zu müssen:

```
Fn1 square = new Fn1() {
  public Object call(Object arg) {
    return (Integer) arg * (Integer) arg;
  }
};
```

Damit würde unser Quellcode von oben auch ohne IDE-Unterstützung sehr leserlich:

```
sequence(1,2,3,4,5).map(λ( n, n * n) );
```

Unsere Selektion der VIP-Kunden sähe damit dann so aus:

```
Sequence vipKunden = kundenliste.filter(
  λ(obj, ((Kunde) obj).istVIPKunde() ) );
```

Da die Parameter innerhalb des Lambdas über statische Imports eingebunden werden, sind dies keine einfachen Bezeichner, sondern referenzieren statische Attribute innerhalb der Enumerable-Parameters-Klasse. Das Framework sorgt bei der Bytecode-Manipulation dafür, dass hier jedes Lambda seine eigenen Parameter erhält.

Da aber Enumerable nichts von unserem Code weiß, gibt es dort auch keine entsprechend typisierten Parameter. Wir müssen daher den allgemeinen Parameter **obj** vom Typ **java.lang.Object** verwenden und selbst casten.

Die Verwendung eigener Attribute funktioniert nicht ohne Framework-Anpassungen, da die erforderliche Bytecode-Manipulation nur auf den von Enumerable deklarierten Parametern funktioniert.

Zusammenfassung

Totally Lazy ist eine kleine Bibliothek, die ohne zusätzliche Abhängigkeiten eingebunden werden kann. Sie bietet eine Reihe von Callable-Klassen an, die gängige Funktionen implementieren. Mit umfangreichen Higher-Order-Functions, die man so aus Sprachen wie ML kennt, ist die Verarbeitung von Collections sehr natürlich. Die funktionalen Elemente lassen sich leicht mit anderen Programmteilen kombinieren, ohne dabei unnatürlich zu wirken.

Der Name ist Programm. Sequenzen werden erst verarbeitet, wenn die Ergebnisse benötigt werden. Das heißt, wenn Werte aggregiert werden oder aber ausgegeben werden sollen. Darüber hinaus können Entwicklungsumgebungen anonyme innere Klassen verkürzt darstellen, sodass man meint, Java könne bereits heute Lambdas.

Wer dann auch noch Lambdas verwenden möchte, kann mit der Bibliothek Enumerable ebenfalls dafür Unterstützung bekommen. Dies funktioniert dann aber nur mit Bytecode-Manipulation – entweder zur Laufzeit beim Laden der Klassen per Instrumentierung oder aber per Bytecode-Manipulation zur Build-Zeit.

Literatur und Links

[enumerable] `https://github.com/hraberg/enumerable`
[total] `http://code.google.com/p/totallylazy/`
[Wamp11] D. Wampler, Functional Programming for Java Developers, O'Reilly, 2011

Phillip Ghadir baut am liebsten tragfähige, langlebige Softwaresysteme. Er ist Mitglied der Geschäftsleitung bei innoQ und hat sich früh auf Architekturen für verteilte, unternehmenskritische Systeme spezialisiert. Darüber hinaus ist er Mitbegründer und aktives Mitglied des iSAQB, des International Software Architecture Qualification Board.
E-Mail: phillip.ghadir@innoq.com