Java aktuell

iJUG Verbund
www.ijug.eu

Cloud
Migration in die Cloud,
Helm Charts

Cloud Native
Cloud Native und AI
in Unternehmen

Java
JavaFinder, Eclipse Starter,
Distributed Message Schema

Ausgabe 06/2023

Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler

D: 4,90 EUR | CH: 9,80 CHF | A: 5,60 EUR | Benelux: 5,80 EUR

4 191978 304903 06

Unbekannte Kostbarkeiten des SDK Heute: Die Klasse ClassValue

Bernd Müller, Ostfalia
heute mit Gastautor Markus Karg, Head Crashing Informatics

Das Java SDK enthält eine Reihe von Features, die wenig bekannt sind. Wären sie bekannt und würden sie verwendet, könnten Entwickler viel Arbeit und manchmal sogar zusätzliche Frameworks einsparen. Wir wollen in dieser Reihe derartige Features des SDK vorstellen: die unbekannten Kostbarkeiten.

Die Klasse `ClassValue` wurde mit Java 7 in das JDK aufgenommen. Sie ist definitiv eine unbekannte Kostbarkeit und beschreibt sich selbst als „*Lazily associate a computed value with (potentially) every type*" *[1]*. Diese recht abstrakte Beschreibung – mit nicht sofort ins Auge fallenden sinnvollen Verwendungsmöglichkeiten – ist sicher mit ein Grund für den geringen Bekanntheitsgrad. Die Anzahl möglicher Anwendungsfälle ist unserer Meinung nach überschaubar, aber sie existieren. In dieser Ausgabe der unbekannten Kostbarkeiten werden wir zwei Beispiele der Verwendung von `ClassValue` vorstellen. „Wir" sind dieses Mal Markus Karg, dem regelmäßigen Java-aktuell-Leser als Autor der Eclipse Corner bekannt, und Bernd Müller. Die Idee und die maßgeblichen Arbeiten am Artikel gehen auf Markus zurück.

Die Verwendungsmöglichkeiten der Klasse ClassValue

Den ersten Satz zur Beschreibung der Klasse `ClassValue` im Package `java.lang` haben wir oben schon wiedergegeben. Hier nun der volle Wortlaut: „*Lazily associate a computed value with (potentially) every type. For example, if a dynamic language needs to construct a message dispatch table for each class encountered at a message send call site, it can use a ClassValue to cache information needed to perform the message send quickly, for each class encountered.*" *[1]*

Man sieht, dass der Autor der Klasse dynamische Sprachen, beziehungsweise deren Implementierungen, als potenzielle Anwendungsfälle sieht. In der Tat zeigt eine kurze Internet-Recherche, dass zum Beispiel Scala und Groovy die Klasse verwenden. Aber auch in der Java-Welt gibt es manchmal die Anforderung, (beliebige) Informationen an (beliebige) Typen (hier: Klassen) zu binden. Dies soll an zwei Beispielen konkretisiert werden: einem On-Demand-Hash-Code und dem Augmentieren von Business-Objekten mit Technikdetails.

On-Demand-Hash-Code

Im OpenJDK-Projekt Lilliput *[2]* wird daran gearbeitet, die Größe von Objekt-Headern und anderen Objektverwaltungsinformationen in der JVM zu minimieren. Eine Motivation hierfür ist, dass weniger Zeit in die Garbage Collection investiert werden muss, da insgesamt mehr Java-Objekte im gleichen Speicherbereich unterzubringen sind. Ein dazu diskutierter Ansatz ist der Verzicht auf einige Meta-Informationen eines jeden Java-Objektes, beispielsweise des Identitäts-Hash-Codes, also jenes statischen Wertes, den die Methode `System.identityHashCode(Object)` liefert. Dieser Wert wird als weitgehend ungenutzt betrachtet, aber trotzdem berechnet und die JVM muss ihn für jede Instanz jeder Klasse speichern. In Summe benötigt sie dazu erhebliche Mengen an Speicher.

Könnte man also auf diese 32 Bit pro Java-Objektinstanz verzichten oder zumindest nur dann Speicher für den Hash-Code belegen, wenn das betreffende Objekt auch wirklich nach seinem Hash-Code gefragt wird? Und viel wichtiger: Wie kann man ein Feld bei Bedarf zur Lauf-

zeit an eine Klasse respektive eine Objektinstanz anfügen? Tatsächlich geht das über den Umweg eines `ClassValue`, wie der folgende Quellcode am Beispiel eines hypothetischen Umbaus von `System.identityHashCode(Object)` demonstriert *(siehe Listing 1)*.

```
public static int identityHashCode(Object o) {
  return HashCodeCache.get(o);
}
```

Listing 1

Die verwendete Klasse `HashCodeCache` ist in *Listing 2* abgebildet. Der Code ist vollständig funktional, aber aus didaktischen Gründen stark vereinfacht. Das OpenJDK-Team würde zur Vermeidung der Boxing-/Unboxing-Kosten definitiv die generische `WeakHashMap` durch eine auf den Anwendungszweck optimierte `IntHashMap` ersetzen.

Das Beispiel ist verblüffend kurz, aber effektiv, und besteht genau genommen nur aus zwei Zeilen des zur Laufzeit ausgeführten, prozeduralen Codes. Die ganze Magie steckt in den JRE-Bestandteilen `ClassValue` und `WeakHashMap`. Wie man sieht, wird keineswegs wirklich dynamisch ein Feld an ein Objekt angefügt, sondern unter Ignoranz jeglicher Objektorientierung der virtuellen Maschine erklärt, sie möge sich über zwei Maps behelfen – eine hängt den `ClassValue` (hier: die zweite Map) an die Klasse, die zweite merkt sich (ohne die Garbage Collection zu verhindern) den ersten jemals berechneten Hash-Code einer Objektinstanz. In der Summe wird nur Speicher für jene Klassen und Instanzen verbraucht, deren Hash-Code mindestens einmal benötigt wurde, die noch referenziert werden, und somit noch nicht durch den Garbage Collector entsorgt werden können.

Ganz nebenbei bemerkt: Ab einer bestimmten Komplexität der Hash-Methode, beziehungsweise Anzahl und Typen der beteiligten Variablen, kann die Verwendung dieses Cache auch nützlich für die Implementierung der Methode `Object.hashCode()` in Projekten des Lesers sein; ein Microbenchmark mit JMH sollte Aufschluss bringen, wann der Break-Even-Point erreicht ist. Sofern jedoch von vorneherein feststeht, dass die eigene Klasse auf jeden Fall einmal in einer Collection landen wird, ist es sinnvoll, gleich einen vorberechneten Hash-Code in einem gewöhnlichen internen Feld zu speichern. Der Mehraufwand für den `ClassValue` lohnt nur dann, wenn diese Verwendungsform nicht bekannt oder zumindest unwahrscheinlich ist.

Augmentieren von Business-Objekten

Kommen wir zum zweiten Beispiel. Markus ist stets auf Separation of Concerns und verständlichen Code bedacht und sah sich mit einem simplen Problem konfrontiert: Der Zustand eines Geschäftsobjekts sollte über ein externes API (zum Beispiel HTTP) mit einem Schlüssel adressiert werden können, der sich nicht aus dem Zustand des Objekts ableiten lässt, da er rein durch die Speicherform, also technisch bedingt ist, und nicht Teil des Domänenmodells ist. Unter Informatikern: Es kommt ein nicht-natürlicher Primärschlüssel zum Einsatz, zum Beispiel eine vom Speichersystem automatisch vergebene UUID. Wie jedoch kann man bei Auflistung aller gespeicherten Objekte den Code sauber, also frei von rein technischen Sachverhalten halten, das heißt, eine Liste rein mit den Geschäftsdaten zurückgeben, ohne explizit auch die technisch ja durchaus benötigten Schlüssel stets mitzugeben?

Schauen wir uns die Ausgangslage, also den unsauberen Code, in Form einer klassischen JAX-RS-Anwendung an, der das Ganze mittels Map löst *(siehe Listing 3)*.

```
public record Book (String isbn, String title) {}

@Inject
private BookStore store;

@GET
@Path("books")
public Map<UUID, Book> list() {
  return store.list();
}
```

Listing 3

Tatsächlich widerspricht es modernen Software-Engineering-Grundsätzen, dass Business-Modell mit technischen Sachverhalten zu verunreinigen. Dass ein Buch aus internen, technischen Gründen des `BookStore` einen Schlüssel in Form einer UUID besitzt, sollte in der JAX-RS-Ressource nicht zu erkennen sein. Schöner wäre es, dies der Technologie-Anpassungsebene, im JAX-RS-Terminus einem `MessageBodyWriter` oder `WriterInterceptor`, zu überlassen, wodurch der Code wieder rein business-orientiert wird *(siehe Listing 4)*.

```
private static class HashCodeCache extends ClassValue<Map<Object, Integer>> {

  private static final HashCodeCache SINGLETON = new HashCodeCache();

  private HashCodeCache() {};

  static int get(Object o) {
     return SINGLETON.get(o.getClass()).computeIfAbsent(o, Object::hashCode);
  }

  @Override
  protected Map<Object, Integer> computeValue(Class<?> ignored) {
     return new WeakHashMap<Object, Integer>();
  }
}
```

Listing 2

```
@GET
@Path("books")
public List<Book> list() {
  return store.list();
}
```

Listing 4

Die störende UUID ist nun zwar verschwunden, doch auf HTTP-Ebene ist es weiterhin zwingend notwendig, dem Aufrufer einen URI mitzuteilen, der eben jene UUID enthält. Woher aber soll der `MessageBodyWriter` beim Rendern des JSON-Ergebnisses wissen, wie die UUID jedes Buches lautet? Die Lösung liegt auch in diesem Fall in einem `ClassValue`. Die in *Listing 5* dargestellte Klasse ID erbt von `ClassValue` und realisiert dies. Die Verwendung der Klassen Book und UUID dienen der besseren Verständlichkeit im Rahmen dieses Artikels. Der Produktiv-Code verwendet ausschließlich die Klasse `Object`, da die tatsächlichen Datentypen von Payload (JAX-RS Entity) und Adresse (JAX-RS URI) für Produzenten und Konsumenten und ebenso für den `ClassValue` irrelevant sind.
Im Unterschied zum ersten Beispiel wird der zugeordnete Wert hier nicht durch den `ClassValue` selbst berechnet, sondern explizit über die Methode `ID.assign(Book, UUID)` zugewiesen. Dies erfolgt durch den Produzenten der Adresse, in diesem Fall also den `BookStore`. Das Auslesen geschieht in einem `WriterInterceptor` per `ID.of(Book)`, der dann per `setEntity()` jedes Buch durch einen `Map.Entry<Book,UUID>` ersetzt. Der `MessageBodyWriter` erhält somit also wieder eben jene Kombination, nur nicht von der Business-Ebene (Ressource), sondern von der Technik-Ebene (`WriterInterceptor`). Aus Platzgründen, und da es thematisch zu weit abschweift, verzichten wir auf die Darstellung des `MessageBodyWriter` und `WriterInterceptor`.

```
public class ID extends ClassValue<Map<Book, UUID>> {

  private static final ID SINGLETON = new ID();

  private ID() {};

  public static void assign(Book book, UUID id) {
     SINGLETON.get(Book.class).put(book, id);
  }

  public static UUID of(Book book) {
     return SINGLETON.get(Book.class).get(book);
  }

  @Override
  protected Map<Book, UUID> computeValue(Class<?> ignored) {
     return new WeakHashMap<Book, UUID>();
  }
}
```

Listing 5

Wieso benutzen wir nicht einfach direkt einen `WeakHashMap`-Singleton statt eines `ClassValue`-Singletons? Der Grund ist, dass wir uns hier in einer Jakarta-EE-Umgebung bewegen, somit für Produzenten und Konsumenten von Buch-IDs separate Classloader-Hierarchien Anwendung finden könnten, die im schlimmsten Fall nichts gemeinsam haben außer der Java-Runtime. Entsprechend würden zwei Singletons existieren, einer, in den geschrieben werden würde, und einer, aus dem gelesen werden würde, womit die Kommunikation nicht gewährleistet wäre. Gerade für Jakarta-EE-Einsteiger eine schwer zu erkennende Fehlerquelle.

Zusammenfassung

Die Klasse `ClassValue` erlaubt es, praktisch beliebige Informationen an eine Klasse zu binden. Ihre Verwendung empfiehlt sich, wenn ein Attribut einer Klasse nur bei sehr wenigen Instanzen von seinem Initialwert abweicht oder, wenn ein zusätzliches Attribut benötigt wird, das aber nicht zu einer existierenden Klasse hinzugefügt werden kann oder soll. Im ersten Fall, der Abweichung vom Initialwert, wird Speicherplatz gespart. Der zweite Fall, das Hinzufügen eines Attributs, wird praktiziert, wenn der Quell-Code nicht vorliegt, oder aus methodischen Gründen, im Beispiel etwa Separation of Concerns, wenn die Code-Qualität erhöht werden soll.

Referenzen

[1] *https://docs.oracle.com/javase/7/docs/api//java/lang/ClassValue.html*

[2] *Lilliput. https://openjdk.org/projects/lilliput/*

Markus Karg

markus@headcrashing.eu

Markus Karg ist Entwicklungsleiter eines mittelständischen Softwarehauses sowie Autor, Konferenzsprecher und Consultant. JAX-RS hat der Sprecher der Java User Group Goldstadt von Anfang an mitgestaltet, zunächst als freier Contributor, seit JAX-RS 2.0 als Mitglied der Expert Groups JSR 339 und JSR 370.

Bernd Müller

Ostfalia

bernd.mueller@ostfalia.de

Nach seinem Studium der Informatik und der Promotion arbeitete Bernd Müller für die IBM und die HDI Informationssysteme. Er ist Professor, Geschäftsführer, Autor mehrerer Bücher zu den Themen JSF und JPA, sowie Speaker auf nationalen und internationalen Konferenzen. Er engagiert sich im iJUG und speziell in der JUG Ostfalen.

Mitglieder des iJUG

01 BED-Con e.V.
02 Clojure User Group Düsseldorf
03 DOAG e.V.
04 EuregJUG Maas-Rhine
05 JUG Augsburg
06 JUG Berlin-Brandenburg
07 JUG Bremen
08 JUG Bielefeld
09 JUG Bonn
10 JUG Darmstadt
11 JUG Deutschland e.V.
12 JUG Dortmund
13 JUG Düsseldorf rheinjug
14 JUG Erlangen-Nürnberg
15 JUG Freiburg
16 JUG Goldstadt
17 JUG Görlitz
18 JUG Hannover
19 JUG Hessen
20 JUG HH
21 JUG Ingolstadt e.V.
22 JUG Kaiserslautern
23 JUG Karlsruhe
24 JUG Köln
25 Kotlin User Group Düsseldorf
26 JUG Mainz
27 JUG Mannheim
28 JUG München
29 JUG Münster
30 JUG Oberland
31 JUG Ostfalen
32 JUG Paderborn
33 JUG Saxony
34 JUG Stuttgart e.V.
35 JUG Switzerland
36 JSUG
37 Lightweight JUG München
38 SUG Deutschland e.V.
39 JUG Thüringen
40 JUG Saarland
41 JUG Duisburg

iJUG Verbund
www.ijug.eu

Impressum

Java aktuell wird vom Interessenverbund der Java User Groups e.V. (iJUG) (Tempelhofer Weg 64, 12347 Berlin, *www.ijug.eu*) herausgegeben. Es ist das User-Magazin rund um die Programmiersprache Java im Raum Deutschland, Österreich und Schweiz. Es ist unabhängig von Oracle und vertritt weder direkt noch indirekt deren wirtschaftliche Interessen. Vielmehr vertritt es die Interessen der Anwender an den Themen rund um die Java-Produkte, fördert den Wissensaustausch zwischen den Lesern und informiert über neue Produkte und Technologien.

Java aktuell wird verlegt von der DOAG Dienstleistungen GmbH, Tempelhofer Weg 64, 12347 Berlin, Deutschland, gesetzlich vertreten durch den Geschäftsführer Fried Saacke, deren Unternehmensgegenstand Vereinsmanagement, Veranstaltungsorganisation und Publishing ist.

Die DOAG Deutsche ORACLE-Anwendergruppe e.V. hält 100 Prozent der Stammeinlage der DOAG Dienstleistungen GmbH. Die DOAG Deutsche ORACLE-Anwendergruppe e.V. wird gesetzlich durch den Vorstand vertreten; Vorsitzender: Björn Bröhl. Die DOAG Deutsche ORACLE-Anwendergruppe e.V. informiert kompetent über alle Oracle-Themen, setzt sich für die Interessen der Mitglieder ein und führen einen konstruktiv-kritischen Dialog mit Oracle.

Redaktion:
Sitz: DOAG Dienstleistungen GmbH
ViSdP: Fried Saacke
Redaktionsleitung: Lisa Damerow
Kontakt: redaktion@ijug.eu

Redaktionsbeirat:
Andreas Badelt, Marcus Fihlon, Markus Karg, Manuel Mauky, Bernd Müller, Benjamin Nothdurft, Daniel van Ross, Bennet Schulz

Titel, Gestaltung und Satz:
Alexander Kermas,
DOAG Dienstleistungen GmbH

Bildnachweis:
Titel: Bild © Designed by Freepik
https://freepik.com
S. 12: Bild © vector4stock
https://freepik.com
S. 16 + 17: Bild © Designed by fullvector
https://freepik.com
S. 24 + 25: Bild © Designed by Ddraw
https://freepik.com
S. 34 + 35: Bild © Designed by pch.vector
https://freepik.com
S. 38 + 39: Bild © Designed by rawpixel.com
https://freepik.com
S. 44 + 45: Bild © anttoniart
https://stock.adobe.com
S. 52 + 53: Bild © AI generiert by GBTaylor
https://pixabay.com
S. 60 + 61: Bild © Designed by redgreystock
https://freepik.com
S. 63: Bild © Designed by storyset
https://freepik.com

Anzeigen:
DOAG Dienstleistungen GmbH
Kontakt: sponsoring@doag.org

Mediadaten und Preise:
www.doag.org/go/mediadaten

Druck:
WIRmachenDRUCK GmbH
www.wir-machen-druck.de

Inserentenverzeichnis