Middleware – Cloud Computing – Übung

MapReduce: Implementierungstipps

Wintersemester 2022/23

Laura Lawniczak, Tobias Distler, Ines Messadi

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)
https://sys.cs.fau.de

Lehrstuhl für Verteilte Systeme
und Betriebssysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Überblick

Implementierungstipps

Abstract Factory Entwurfsmuster

Framework-Entwicklung

- Framework stellt **Rahmen** für Anwendungen auf
  - Lediglich **grundsätzlicher Ablauf** vorgegeben
  - Details der Anwendung nicht vorab bekannt
    $\rightarrow$ Hohe Flexibilität und Konfigurierbarkeit notwendig

- Im Fall des MapReduce-Frameworks in der Aufgabe:
  - Deserialisierung
  - Mapper
  - Reducer
  - Sortierkriterium

- Auswählbare Implementierung für einzelne Schritte
  - Framework muss notwendige Objekte selbst instanziieren

  $\Rightarrow$ Lösung mittels „Factory Pattern“

Factory Pattern

- Problemstellung: Es sollen Objekte instanziiert werden, welche eine bestimmte Schnittstelle zur Verfügung stellen, ohne dass der genaue Typ vorab bekannt ist.

  → **Kapselung der Instanziierung** in eigener Klasse

- Beispiel:

```
public class WordCountMapper implements Mapper {
  ...
}

public class WordCountFactory {
  public Mapper createMapper() {
    return new WordCountMapper();
  }
}
```

- **Allerdings**: Klasse `WordCountFactory` muss Framework bekannt sein

Abstract Factory Pattern

- Lösung durch weitere Abstraktionsschicht: Schnittstelle zur Instanziierung

```
public class WordCountMapper implements Mapper { ... }

public interface MapperFactory {
  public Mapper createMapper();
}

public class WordCountFactory implements MapperFactory {
  public Mapper createMapper() {
    return new WordCountMapper();
  }
}
```

- Verwendung:

```
void myMethod(MapperFactory mfact) {
  Mapper m = mfact.createMapper();
  ...
}
```

Implementierungstipps

Vergleichen und Sortieren mit Java

Sortieren mittels Comparator-Objekten

- **Comparable**
  - Vergleicht Objekt mit **anderem gegebenen Objekt**

```
public interface Comparable<T> {
  public int compareTo(T o);
}
```

- **Comparator**
  - Vergleicht **zwei gegebene Objekte miteinander**
  - **Invariante:** `Object.equals(o1, o2) == true` $\Leftrightarrow$ `Comparator.compare(o1, o2) == 0`
  - **Kein Ableiten** der zu sortierenden Objekte notwendig

```
public interface Comparator<T> {
  public int compare(T o1, T o2);
}
```

- Verwendung:
  - Methoden `compareTo()` und `compare()` liefern Integer zurück
    - **negativ**: Linker Wert **kleiner** als rechter Wert (kommt **vor**...)
    - **0**: Beide Werte sind **gleich** (**äquivalent**)
    - **positiv**: Linker Wert **größer** als rechter Wert (kommt **nach**...)

```
int x = links.compareTo(rechts);
int y = comparator.compare(links, rechts);
```

- Beispiel: Strings in einer TreeMap rückwärts sortieren

```
class RevStringComparator implements Comparator<String> {
  public int compare(String o1, String o2) {
    return -o1.compareTo(o2);
  }
}
```

```
RevStringComparator revcmp = new RevStringComparator();
TreeMap<String,X> treemap = new TreeMap<String,X>(revcmp);
```

Implementierungstipps

Zusammenführung vorsortierter Listen

Zusammenführung mittels Priority-Queues

- Aufgabe: Zusammenführen bereits vorsortierter Listen
  - Vergleich des obersten Elements über alle Listen
  - Kleinstes Element bestimmt nächstes Ausgabeelement

- Datenstruktur **Priority-Queue**
  - Einfügen von Elementen mit zugeordneter Priorität
  - Entfernen entnimmt immer Element mit **höchster** Priorität
  - Üblicherweise als Heap-Datenstruktur implementiert

```
public PriorityQueue(int capacity, Comparator c); // Festlegen der Sortierung mittels Comparator
public boolean add(E item);                       // Einfügen eines Elements vom Typ E
public E peek();                                  // Abfrage des obersten Elements
public E poll();                                  // Entnahme des obersten Elements
```

- Nutzung als Merge-Algorithmus
  - Priorität entspricht Wertigkeit des **obersten Elements** jeder **Liste**
  - Entnahme aus Priority-Queue liefert Liste mit nächstem Element

1. Priority-Queue mit vorsortierten Listen befüllen
2. Entnahme des Elements höchster Priorität liefert Liste, welche das nächste auszugebende Listenelement an erster Stelle enthält
3. Ausgeben und Entfernen des obersten Listenelements aus der entnommenen Liste
4. Liste wieder in Priority-Queue einfügen
5. Wiederholen ab (2), bis alle Listen leer sind

Implementierungstipps

Futures

Futures

- Allgemeine Schnittstelle

```
boolean isDone();
<beliebiger Datentyp> get();
```

- Funktionsweise

1. Beim asynchronen Aufruf wird (statt dem eigentlichen Ergebnis) sofort ein Future-Objekt zurückgegeben
2. Das Future-Objekt lässt sich befragen, ob der tatsächliche Rückgabewert der Operation bereits vorliegt bzw. ob die Operation beendet ist → `isDone()`
3. Ein Aufruf von `get()`
   - liefert das Ergebnis der Operation sofort zurück, sofern es zu diesem Zeitpunkt bereits vorliegt **oder**
   - blockiert solange, bis das Ergebnis eingetroffen ist

- Schnittstelle **Future**

```
public interface Future<V> {
  public V get()  throws InterruptedException, ExecutionException;
  public V get(long timeout, TimeUnit unit)
    throws InterruptedException, ExecutionException, TimeoutException;

  public boolean isDone();

  public boolean cancel(boolean mayInterruptIfRunning);
  public boolean isCancelled();
}
```

- Umfang
  - Methoden der allgemeinen Future-Schnittstelle
  - Zusätzliche Methoden zum Abbrechen von Tasks
  - `get()` wirft ggf. von Operation geworfene `Exception`
    - Verpackt als `ExecutionException`
    - Zugriff auf ursprüngliche Exception: `excecutionException.getCause()`

- Interface `ExecutorService`
  - Erlaubt asynchrone Ausführung von Tasks
  - Task bei Executor-Service „abgeben“, Ergebnis per Future

```
// ExecutorService erstellen (Beispiele)
public static ExecutorService newSingleThreadExecutor(); // einzelner Thread
public static ExecutorService newFixedThreadPool(int nThreads); // konstante Anzahl Threads

<T> Future<T> submit(Callable<T> task);
Future<?> submit(Runnable task)

public void shutdown();
boolean awaitTermination(long timeout, TimeUnit unit);
```

- Interface `Callable`

```
public interface Callable<V> {
  V call() throws Exception;
}
```

- Interface `Runnable`

```
public interface Runnable {
  void run();
}
```

- Beispielklasse

```
public class FutureExample implements Callable<Integer> {
  private int a;

  public FutureExample(int a) {
    this.a = a;
  }
  public Integer call() throws Exception {
    return a * a;
  }
}
```

- Aufruf

```
ExecutorService es = Executors.newSingleThreadExecutor();
FutureExample task = new FutureExample(4);
Future<Integer> f = es.submit(task);
[...]
try {
  System.out.println("result: " + f.get());
} catch (InterruptedException | ExecutionException e) {
  // Fehlerbehandlung
}
```

Implementierungstipps

Daten finden und extrahieren

Extrahieren von Daten

- Typische MapReduce-Anwendung: Extrahieren von Daten
  - Statistiken, Data Mining
  - Mustererkennung, Machine Learning
  - Graph-Algorithmen

- Eingabedaten häufig in Form von Textzeilen
- Zeilenweise Partitionierung von Eingabedaten problematisch:
  Zusammengehörige Daten können in **unterschiedlichen** Worker-Threads verarbeitet werden

- Lösungsmöglichkeiten:
  - Beeinflussung der Partitionierung durch Eingabedaten
  - Verwerfen unvollständiger Datensätze, z. B. statistischen Auswertungen großer Datenmengen

Auffinden und Extrahieren von Zeichenketten

- Einfache Methoden in `Java.lang.String`
- Finden konstanter Zeichenketten
  - Vorwärts suchen ab bestimmter Position:

```
public int indexOf(String str, int start);
```

  - Rückwärts suchen ab bestimmter Position:

```
public int lastIndexOf(String str, int start);
```

- Teilstrings extrahieren:

```
public String substring(int start, int end);
```

  - Ausgabe des Strings ab `start` bis `end`, **ohne** `end` selbst
  - Tipp: Zum Testen ein Zeichen vor und nach dem gesuchten Teilbereich ausgeben lassen