KAFKA-CONNECT – DRITTSYSTEME AN KAFKA ANBINDEN

Allgemein (https://javapro.io/category/allgemein/) and Frameworks & APIs (https://javapro.io/category/frameworks/) 22. Dezember 2020
Streaming (https://javapro.io/tag/streaming/)

Kafka ist eine verteilte Streaming-Plattform die verschiedene Nutzungsszenarien unterstützt. Sie kann als Messaging- oder Speichersystem eingesetzt werden und Datenströme transformieren. Dieser Artikel beschreibt, wie Sie mit Kafka-Connect Drittsysteme wie Redis an Kafka anbinden können.

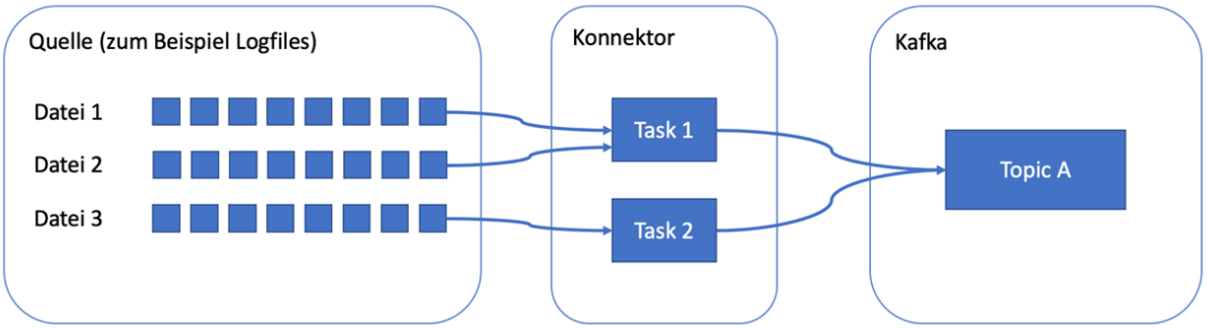
Kafka bietet eine flexible Grundlage, um darauf komplexe Datenverarbeitungsarchitekturen zu bauen. Eine wichtige Frage bei diesen Architekturen ist, wie die Daten in die Plattform gelangen und wie sie diese wieder verlassen. Hier kommt Kafka-Connect ins Spiel, ebenfalls eine quelloffene Komponente von Kafka. Kafka-Connect ist ein Framework mit dem man externe Systeme, wie zum Beispiel Datenbanken, Key-Value-Speicher, Suchindizes oder Dateisysteme an Kafka anschließen kann. Für Kafka-Connect gibt es bereits zahlreiche vorgefertigte Konnektoren, wie zum Beispiel für Splunk oder **Elasticsearch (https://bit.ly/kafka-k1)**[1]. Diese Konnektoren teilen sich in zwei Gruppen auf: Sources und Sinks -also Quellen und Senken. Quellkonnektoren können zum Beispiel ganze Datenbanken auslesen und Änderungen in der Datenbank dann in Kafka-Topics veröffentlichen. Ein anderer Anwendungsfall ist das Sammeln von Metriken von Microservices, um sie dann dem Stream-Processing mit geringer Latenz zur Verfügung zu stellen und sie dann auswerten zu können. Sinks bringen die Daten aus den Kafka-Topics dann zum Beispiel in Suchindizes wie Elasticsearch oder in Batch-Systeme wie Hadoop, zur späteren Offline-Verarbeitung.

Nun könnte man meinen, das alles kann auch ohne ein zusätzliches Framework realisiert werden, indem man über das API von Kafka die Daten nach Kafka schreibt oder daraus liest. Worin liegen also die Vorteile von Kafka-Connect? Kafka-Connect stellt eine API bereit, die es ermöglicht, dass ein Konnektor nach einem Fehler an der Stelle fortfährt, an der er unterbrochen worden ist. Dabei wird der Konnektor automatisch neu gestartet und gegebenenfalls auch auf einen neuen Knoten innerhalb des Kafka-Clusters gestartet. Zusätzlich macht es die API einfacher mehrere Worker parallel zu starten, die dasselbe Quell- bzw. Zielsystem versorgen.

Auch wenn bereits viele verschiedene Konnektoren verfügbar sind, kann es dennoch vorkommen, dass für ein Legacy-System oder eine Eigenentwicklung kein Konnektor zur Verfügung steht oder dass die benötige Funktionalität nicht angeboten wird. Das stellt jedoch kein Hindernis für den Einsatz von Kafka und Kafka-Connect dar. Das Implementieren eines eigenen Konnektors ist relativ einfach und ermöglicht es praktisch jedes System mit Kafka zu verbinden. Wie bereits erwähnt unterscheidet Kafka-Connect zwischen Quellen und Senken, je nachdem ob Daten nach Kafka hinein oder hinaus bewegt werden. Es gibt also zwei verschiedene Interfaces, die je nach Anwendungsfall implementiert werden müssen. Um eine Quelle anzuschließen müssen

org.apache.kafka.connect.source.SourceConnector und **org.apache.kafka.connect.source.SourceTask**

implementiert werden. Für eine Senke sind es **SinkConnector** und **SinkTask**. Ein **JDBCSourceConnector** würde zum Beispiel die Daten einer JDBC-kompatiblen Datenbank nach Kafka importieren und ein **HDFSSinkConnector** würde den Inhalt eines Kafka-Topics nach **HDFS** exportieren. Dabei ist zu beachten, dass der Konnektor selbst nicht für das Kopieren der Daten verantwortlich ist. Er erhält beim Starten von Kafka-Connect seine Konfiguration und leitet daraus eine Konfiguration für die Tasks ab. Die Konfiguration des Konnektors enthält zum Beispiel den Dateipfad oder die JDBC-URL und wie viele Tasks erzeugt werden sollen. Bei einem Dateikonnektor könnte zum Beispiel konfiguriert werden, dass es zwei Tasks geben darf. Der Konnektor wäre dann dafür verantwortlich, nicht nur den Dateipfad an den Task zu übergeben, sondern auch welcher Bereich aus der Datei gelesen werden soll, so dass es zwei Tasks geben kann, die sich nicht gegenseitig beeinflussen. Diese Tasks werden dann von Kafka-Connect-Workern **verarbeitet (https://bit.ly/kafka-k2)**[2]. Auch bei diesen Tasks wird wieder zwischen Quelle und Senke unterschieden. Der Konnektor kann bei Bedarf auf Änderungen des externen Systems reagieren und den Task geeignet rekonfigurieren.



Schemabild wie ein Quellkonnektor mit Kafka zusammenarbeitet (Abb. 1)

Hat der Task nun die Konfiguration eines Konnektors bekommen, ist dieser dafür verantwortlich die Daten in das konfigurierte Kafka-Topic zu schreiben oder aus einem Topic zu lesen. Dabei müssen die Daten ebenfalls in verschiedene Partitionen aufgeteilt werden, analog dazu, wie Kafka Topics strukturiert. Innerhalb jeder Partition sind die Daten wieder eine geordnete Sequenz mit entsprechenden Offsets. Diese Aufteilung zu finden kann je nach Anwendung sehr einfach oder aber aufwändig sein. Wenn zum Beispiel Log-Files **(Abb. 1)** gelesen werden, kann jede einzelne Datei einer Partition entsprechen, jede Zeile ist ein Eintrag und der Offset ist einfach die Position innerhalb der Datei. Bei einem JDBC-Konnektor kann es da schon schwieriger werden. Eine Möglichkeit wäre, jede Tabelle auf ein Topic abzubilden. Innerhalb des Topics kann dann der Zeitstempel der letzten Abfrage genutzt werden, um eine Reihenfolge zu definieren.

Redis-Konnektor

Wie wird nun also ein Konnektor konkret implementiert? Als Quellsystem soll ein Redis-Cache dienen, der über das Pub/Sub-System angeschlossen wird. Das ermöglicht es zu zeigen, welche Schnittstellen für Kafka-Connect implementiert werden müssen und welche Funktionen dadurch nutzbar werden, ohne dass die Komplexität zu groß wird. Damit sich der Task mit Redis verbinden kann, benötigen wir Informationen wie Redis erreichbar ist. Diese Informationen werden von Kafka-Connect an den Redis-Konnektor weitergegeben **(Listing 1)**.

Es sind also folgende Eigenschaften notwendig: Hostname Dieses Beispiel ist stark vereinfacht, um die wesentlichen Implementierungsschritte zu zeigen. Ein Redis-Konnektor mit vollem Funktionsumfang würde nicht nur einen einzelnen Channel beobachten, sondern mit Hilfe von **PUBSUB CHANNELS** alle aktiven Channels von Redis abfragen und für jeden Channel dann einen Task erzeugen, der dann in sein eigenes Topic schreibt. Eines der Designziele von Kafka-Connect ist, dass die Konnektoren mit großen Mengen an Daten umgehen können, bei gleichzeitig einfacher Konfiguration.

(Listing 1)

```
1   public class RedisSourceConnector extends SourceConnector {
2       private String topic;
3       private String host;
4       private int port;
5       private String channel;
6   }
```

KafkaConnect.java view raw (https://gist.github.com/javapro-magazine/a558b68a01f8c52341e0bfe9364771f5/raw/caba4d96dd433e4d4638a64738a5be0e5bf79c29/KafkaConnect.java) (https://gist.github.com/javapro-magazine/a558b68a01f8c52341e0bfe9364771f5#file-kafkaconnect-java) hosted with ❤ by GitHub (https://github.com)

Damit diese Konfigurationsparameter beim Anlegen eines Konnektors auch übergeben werden können, muss die **start** Methode geeignet überschrieben werden **(Listing 2)**. Diese Methode bekommt eine **Map** aus der dann die Konfigurationseinstellungen für den Redis-Konnektor extrahiert werden.

(Listing 2)

```
1   @Override
2   public void start(final Map<String, String> props) {
3     final AbstractConfig parsedConfig = new AbstractConfig(CONFIG_DEF, props);
4     topic = parsedConfig.getString(TOPIC_CONFIG);
5     host = parsedConfig.getString(HOST);
6     port = parsedConfig.getInt(PORT);
7     channel = parsedConfig.getString(CHANNEL)
8   }
```

KafkaConnect2.java view raw (https://gist.github.com/javapro-magazine/76229bf6af7553a482bfad05cc570e88/raw/d1ac3c11635f8245d23607e190953cdbef39b65a/KafkaConnect2.java) (https://gist.github.com/javapro-magazine/76229bf6af7553a482bfad05cc570e88#file-kafkaconnect2-java) hosted with ❤ by GitHub (https://github.com)

Die generische **Map** wird zunächst mit Hilfe einer Konfigurationsdefinition **CONFIG_DEF** in eine **AbstractConfig** umgewandelt, anschließend werden dann die einzelnen Konfigurationsparameter extrahiert. Die **CONFIG_DEF** kann im veröffentlichten **Quellcode (https://bit.ly/quellcode)**[3] eingesehen werden. Diese Konfigurationsdefinition wird zum Beispiel von dem Confluent-Control-Center verwendet, um eine Konfigurationsoberfläche für den Konnektor anzuzeigen.

Redis-Task

Damit Kafka-Connect nun einen **Source-Task** erstellen kann, müssen noch zwei weitere Methoden überschrieben werden. Zum einen muss mitgeteilt werden, welche Klasse den Task für diesen Konnektor übernimmt und wie dieser Task konfiguriert werden muss. Man erkennt, dass in dem **RedisSourceConnector** in erster Linie die Konfiguration für die Tasks aufbereitet wird, um dann zu entscheiden wie viele Tasks erzeugt werden sollen. Um den Konnektor robuster zu machen, könnte man hier eine Validierung einbauen und prüfen ob mit den übergebenen Parametern wie **HOST** und **PORT** Redis überhaupt erreichbar ist.

Konkret müssen die Methoden **taskClass** und **taskConfigs** überschrieben werden. In **taskClass** wird die Klasse definiert, die den Task dann tatsächlich ausführt, also **RedisSourceTask**. In **taskConfigs** muss die Konfiguration für die Tasks erstellt werden **(Listing 3)**. Je nach Konnektor kann dies mehr oder weniger aufwendig sein. In diesem einfachen Beispiel wird die Konfiguration 1:1 an den Task weitergegeben. Das bedeutet auch, dass genau ein Task erzeugt wird. Falls der Anwendungsfall es zulässt, dass mehr als ein Task erzeugt werden kann muss darauf geachtet werden, dass nicht mehr als im Parameter von **taskConfigs** angegeben erzeugt werden.

(Listing 3)

```
1   @Override
2   public List<Map<String, String>> taskConfigs(final int maxTasks) {
3     final ArrayList<Map<String, String>> configs = new ArrayList<>();
4     final Map<String, String> config = new HashMap<>();
5     config.put(HOST, host);
6     config.put(TOPIC_CONFIG, topic);
7     config.put(PORT, String.valueOf(port));
8     config.put(CHANNEL, channel);
9     configs.add(config);
10    return configs;
11  }
```

KafkaConnect3.java view raw (https://gist.github.com/javapro-magazine/c735f0448c9aabc51ed3188505ece5bc/raw/f1cf1c418ef1f00d7fc8ce628cd1c79ed6c202bd/KafkaConnect3.java) (https://gist.github.com/javapro-magazine/c735f0448c9aabc51ed3188505ece5bc#file-kafkaconnect3-java) hosted with ❤ by GitHub (https://github.com)

Damit ist der **RedisSourceConnector** fertig implementiert. Der zugehörige **RedisSourceTask** implementiert wie die Daten kopiert werden. Doch betrachten wir zunächst, wie Kafka-Connect mit diesem Task interagiert. Das **SourceTask** Interface definiert die **poll** Methode, die von den Worker-Prozessen von Kafka-Connect wiederholt aufgerufen wird, um eine Liste von Einträgen abzurufen und diese dann in das entsprechende Topic zu schreiben. Dabei ist zu beachten, dass Kafka-Connect für jeden Task einen dedizierten Thread zur Verfügung stellt. Das bedeutet die Implementierung von **poll** darf blockieren. Das ermöglicht es zunächst mit einer einfachen Implementierung zu starten und diese dann später zu optimieren. Bevor nun Kafka-Connect **poll** aufrufen kann, muss der Task gestartet werden. Dies geschieht indem die **start** Methode aufgerufen wird **(Listing 4)**. Dort wird nun zunächst eine Verbindung zum Redis-Server aufgebaut (1), um dann den **RedisSourceTask** als Listener bei dem Pub/Sub-System von Redis zu registrieren (2).

(Listing 4)

```
1   @Override
2   public void start(final Map<String, String> props) {
3     LOG.info("Start new RedisSourceTask and open connection to Redis.");
4     final String host = props.get(RedisSourceConnector.HOST);
5     final int port = Integer.valueOf(props.get(RedisSourceConnector.PORT));
6     final String channel = props.get(RedisSourceConnector.CHANNEL);
7     topic = props.get(RedisSourceConnector.TOPIC_CONFIG);
8     redisClient = RedisClient.create(RedisURI.Builder.redis(host, port).build());
9     connection = redisClient.client.connectPubSub(); // 1
10    connection.addListener(this); // 2
11    final RedisPubSubCommands<String, String> sync = connection.sync();
12    sync.subscribe(channel); // 3
13    messageQueue = new ConcurrentLinkedDeque<>(); // 4
14  }
```

KafkaConnect4.java view raw (https://gist.github.com/javapro-magazine/8977922c9857c2bd14229d1ac9889477/raw/bc4de8cc286f7842be693df7b2390cedf4610f75/KafkaConnect4.java) (https://gist.github.com/javapro-magazine/8977922c9857c2bd14229d1ac9889477#file-kafkaconnect4-java) hosted with ❤ by GitHub (https://github.com)

Danach wird der konfigurierte Kanal abonniert (3) und eine **Deque** (4) initialisiert **(Listing 4)**. Eine **Deque** ist eine doppelseitige Warteschlange, mit der eine Verbindung zwischen der **poll** Methode und dem **Event-Listener** hergestellt werden kann.

(Listing 5)

```
1   @Override
2   public void message(final String channel, final String message) {
3     LOG.info("Recieved message from redis.");
4     final Map<String, ?> sourcePartition = ImmutableMap.of("channel", channel);
5     final Map<String, ?> sourceOffset = ImmutableMap.of();
6     final SourceRecord record = new SourceRecord(sourcePartition, sourceOffset, topic, SchemaBuilder.STRING_SCHEMA, message); //1
7     messageQueue.add(record); //2
8   }
```

KafkaConnect5.java view raw (https://gist.github.com/javapro-magazine/dbd751b2a7953d370c514a55488b9859/raw/e45fa22f6220884ac4648186f05cef644356137a/KafkaConnect5.java) (https://gist.github.com/javapro-magazine/dbd751b2a7953d370c514a55488b9859#file-kafkaconnect5-java) hosted with ❤ by GitHub (https://github.com)

Da der **RedisSourceTask** als Listener bei dem Redis-Client registriert wurde, muss nun auch das entsprechende Interface implementiert werden **(Listing 5)**. Relevant ist dabei die **message** Methode, die für das Verarbeiten von Nachrichten verantwortlich ist, die aus dem Pub/Sub-System von Redis empfangen werden. Hier wird ein **SourceRecord** erstellt (1) und dann mit **add** an das Ende der Warteschlange angehängt (2). Der **SourceRecord** ist ein von Kafka-Connect zur Verfügung gestellter Datentyp, der einen Eintrag in ein Kafka-Topic beschreibt. Dieser enthält neben den Daten auch das Topic, in das die Daten geschrieben werden sollen und Informationen woher diese Nachricht kam. Die **sourcePartition** beschreibt in diesem Fall den Ursprung der Daten – hier der Kanal, der belauscht wird. Mit dem **sourceOffset** kann angegeben werden, bis zu welcher Stelle man bereits von dem Quellsystem gelesen hat, um bei einem Neustart des Tasks wieder an dieser Stelle fortfahren zu können. Bei dem Redis-Task ist dieser Wert nicht gesetzt, da beim Pub/Sub-System die Nachrichten nicht zwischengespeichert werden, sondern nur an alle Empfänger gesendet werden, die im Moment des Sendens verbunden sind. Wenn eine Datei eingelesen wird, kann zum Beispiel die Zeilennummer als Offset verwendet werden, die gerade gelesen worden ist. Zusätzlich kann noch ein Schema mit übergeben werden, also eine Beschreibung welches Format die Nachricht hat. Dadurch ist es möglich, verschiedene Nachrichten in verschiedenen Formaten an Kafka zu senden und beim Lesen wieder anhand des Schemas zu rekonstruieren.

(Listing 6)

```
1   @Override
2   public List<SourceRecord> poll() throws InterruptedException {
3     final List<SourceRecord> records = new ArrayList<>(1024);
4     int count = 0;
5     SourceRecord record;
6     while (count <= 1024 && null != (record = messageQueue.poll())) { //1
7       records.add(record);
8       count++;
9       LOG.info("Answering poll() with {} records.", count);
10      }
11    return records;
12  }
```

KafkaConnect6.java view raw (https://gist.github.com/javapro-magazine/a86bab08b79149f17be98e601d3edb3e/raw/9071208d1c8edd9c0378f43f58ef379aee47f5d3/KafkaConnect6.java) (https://gist.github.com/javapro-magazine/a86bab08b79149f17be98e601d3edb3e#file-kafkaconnect6-java) hosted with ♥ by GitHub (https://github.com)

Die so erzeugten Einträge in der Warteschlage werden nun durch wiederholtes Aufrufen der **poll** Methode von den Kafka-Connect Workern angeholt und in das angegebene Topic geschrieben. Dabei wird die Warteschlange abgefragt bis sie leer ist und dann an Kafka-Connect zurückgegeben. Falls ein Quell-System angeschlossen wird, das ein Quittieren der gelesenen Nachrichten unterstützt, beispielsweise eine Message-Queue, kann dies in den **commit** oder **commitrecord** Methoden realisiert werden.

Paketierung des Konnektors

Damit ist die Implementierung eines einfachen Quellkonnektors vollständig. Um diesen Konnektor nun in Kafka-Connect zu benutzen, muss ein Archiv erstellt werden, das sowohl die Implementierung enthält als auch alle relevanten Third-Party-Bibliotheken. Dies kann mit dem **Maven-Assembly-Plugin (https://bit.ly/plugin-p4)**[4] erreicht werden **(Listing 7)**. Dazu packt man das Artefakt (1) inklusive seiner Runtime-Dependencies (2) in ein eindeutig benanntes Zip-Archiv.

(Listing 7)

```
1   <assembly xmlns="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.2"
2             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3             xsi:schemaLocation="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.2 http://maven.apache.org/xsd/assembly-1.1.2.xsd">
4       <id>zip</id>
5       <includeBaseDirectory>true</includeBaseDirectory>
6
7       <formats>
8           <format>zip</format>
9       </formats>
10      <files> <!-- 1 -->
11          <file>
12              <source> ${project.build.directory}/${project.artifactId}-${project.version}.jar</source>
13              <outputDirectory>/</outputDirectory>
14          </file>
15      </files>
16      <dependencySets> <!-- 2 -->
17          <dependencySet>
18              <excludes>
19                  <exclude>${project.groupId}:${project.artifactId}:jar:*</exclude>
20              </excludes>
21              <scope>runtime</scope>
22          </dependencySet>
23      </dependencySets>
24  </assembly>
```

KafkaConnect7.java view raw (https://gist.github.com/javapro-magazine/f6314c3e08b1d3ec7f8035cf2462fc6c/raw/df3cc8c8c7f2ea8aeb0d861f320b9031225467f5/KafkaConnect7.java) (https://gist.github.com/javapro-magazine/f6314c3e08b1d3ec7f8035cf2462fc6c#file-kafkaconnect7-java) hosted with ♥ by GitHub (https://github.com)

Das so erzeugte Archiv enthält dann alles notwendige, um den Konnektor zu distribuieren. Die Installation in Kafka-Connect gestaltet sich dann auch sehr einfach. Das Archiv wird entpackt und das neu entstandene Verzeichnis wird in den Plug-in-Pfad von Kafka-Connect kopiert. Beim nächsten Starten steht der Konnektor dann zur Verfügung.

Fazit:

Um große Datenmengen einfach nach Kafka oder aus Kafka heraus zu bewegen, kann man entweder die Kafka-API direkt benutzen oder auf Kafka-Connect zurückgreifen. Kafka-Connect bietet mit seiner API einen wesentlich einfacheren Weg, um mit Kafka zu interagieren und denkt dabei auch schon an Ausfallsicherheit, Fehlertoleranz und Performance. Die API ist leicht verständlich und ermöglicht es sehr schnell Konnektoren für beliebige Drittsysteme zu entwickeln.

[1] https://bit.ly/kafka-k1 (https://bit.ly/kafka-k1)

[2] https://bit.ly/kafka-k2 (https://bit.ly/kafka-k2)

[3] https://bit.ly/quellcode (https://bit.ly/quellcode)

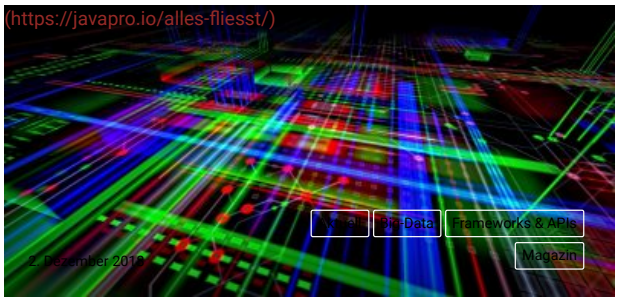
[4] https://bit.ly/plugin-p4 (https://bit.ly/plugin-p4)

Autor - Nikolai Mainiero

Nicolai Mainiero ist Diplom-Informatiker und arbeitet als Software Developer bei der Firma sidion. Er entwickelt seit über zwölf Jahren Geschäftsanwendungen in Java, Kotlin und PHP für unterschiedlichste Kundenprojekte. Dabei setzt er vor allem auf agile Methoden wie Kanban. Außerdem interessiert er sich für funktionale Programmierung, Microservices und reaktive Anwendungen.

Webseite (https://www.sidion.de) | **E-Mail** (mailto:nicolai.mainiero@sidion.de)

Streaming (https://javapro.io/tag/streaming/)



ALLES FLIESST: FAST-DATA-STREAMING UND MESSAGING GEHÖREN ZUSAMMEN (HTTPS://JAVAPRO.IO/ALLES-FLIESST/)

LOAD MORE

Leave a Reply

Name*

Email*

Website*

Post Comment

Autoren

Jetzt Autor werden

Unsere Autoren

Autoren Login

JAVAPRO

Über JAVAPRO

Impressum

Nutzungsbestimmungen

Datenschutz