Apache Kafka em Java

Apache Kafka é uma plataforma de streaming distribuído, projetada para lidar com dados em tempo real com alta resiliência, baixa latência e escalabilidade horizontal.

Ela é amplamente utilizada em sistemas de coleta e processamento de dados, permitindo que grandes volumes de dados sejam consumidos e processados em tempo real.

Kafka funciona como um sistema de mensagens publicador-assinante, onde produtores enviam mensagens para tópicos e consumidores as leem.

Kafka é amplamente usado para:

- **Processamento de fluxo de dados em tempo real:** coletar, armazenar e processar dados em tempo real.
- **Log distribuído:** como uma camada de dados para armazenar logs de sistemas distribuídos.
- **Integração de dados:** atua como um "hub" para integrar diferentes sistemas de dados.

A comunidade Kafka lança atualizações frequentes, com novas funcionalidades e melhorias. Algumas das principais versões incluem:

- **Kafka 1.x:** Primeiras versões com funcionalidades básicas de transmissão de mensagens e persistência.
- **Kafka 2.x:** Inclui melhorias em escalabilidade e estabilidade, adicionando suporte a mais recursos de segurança.
- **Kafka 3.x:** Foca em eficiência e novas funcionalidades, como novas APIs para maior flexibilidade e segurança.

Utilização do Apache Kafka com Java (8, 11 e 17)

Abaixo, veremos como implementar um simples produtor e consumidor Kafka em Java nas versões LTS (8, 11 e 17).

Dependências

Para utilizar o Kafka com Java, será necessário adicionar a dependência do Kafka Client no projeto Maven:

Copiar código

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>3.0.0</version>
</dependency>
```

Código do Produtor Kafka

Aqui, nosso produtor envia uma mensagem ao Kafka para um tópico chamado meu-topico. Vou simular um exemplo de entrada e saída com base nesse código.

```
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.clients.producer.RecordMetadata;

import java.util.Properties;
import java.util.concurrent.ExecutionException;

public class KafkaMessageProducer {

    // Configuração do produtor Kafka
    private static Properties producerProperties() {
        Properties properties = new Properties();
        properties.put("bootstrap.servers", "localhost:9092"); // Endereço do Kafka
        properties.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
        properties.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");
        return properties;
    }

    public static void main(String[] args) {
        // Instância do produtor Kafka
        Producer<String, String> producer = new KafkaProducer<>(producerProperties());

        String topic = "meu-topico";
        String key = "chave-exemplo";
        String value = "Mensagem de teste";

        try {
            // Envia a mensagem para o Kafka de maneira síncrona e aguarda confirmação
            RecordMetadata metadata = producer.send(new ProducerRecord<>(topic, key,
value)).get();
            System.out.printf("Mensagem enviada com sucesso para o tópico %s, partição %d,
offset %d\n",
                    metadata.topic(), metadata.partition(), metadata.offset());
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        } finally {
            // Fecha o produtor para liberar recursos
            producer.close();
        }
    }
}
```

Exemplo de Input e Output do Produtor

- **Input** (Valores fornecidos para o código):
  - Tópico: meu-topico
  - Chave: "chave-exemplo"
  - Valor: "Mensagem de teste"
- **Output** (Saída no console):

Mensagem enviada com sucesso para o tópico meu-topico, partição 0, offset 5

  - **Explicação**: O Kafka retorna o tópico (meu-topico), a partição em que a mensagem foi enviada (0), e o offset (5), indicando a posição da mensagem na fila.

Código do Consumidor Kafka

Agora vamos criar um consumidor Kafka que lê mensagens do tópico meu-topico. Ele exibirá a chave, o valor e o offset das mensagens.

```
import org.apache.kafka.clients.consumer.Consumer;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.clients.consumer.ConsumerRecord;

import java.time.Duration;
import java.util.Collections;
import java.util.Properties;

public class KafkaMessageConsumer {

    // Configuração do consumidor Kafka
    private static Properties consumerProperties() {
        Properties properties = new Properties();
        properties.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092"); //
Endereço do Kafka
        properties.put(ConsumerConfig.GROUP_ID_CONFIG, "meu-grupo"); // Grupo de
consumidores
        properties.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
"org.apache.kafka.common.serialization.StringDeserializer");
        properties.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
"org.apache.kafka.common.serialization.StringDeserializer");
        return properties;
    }

    public static void main(String[] args) {
        // Instância do consumidor Kafka
        Consumer<String, String> consumer = new KafkaConsumer<>(consumerProperties());

        String topic = "meu-topico";

        // Inscreve-se no tópico
```

```
        consumer.subscribe(Collections.singletonList(topic));

        try {
            while (true) {
                // Busca mensagens no tópico
                ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));

                for (ConsumerRecord<String, String> record : records) {
                    System.out.printf("Consumido mensagem: chave=%s, valor=%s, offset=%d\n",
                            record.key(), record.value(), record.offset());
                }
            }
        } finally {
            // Fecha o consumidor para liberar recursos
            consumer.close();
        }
    }
}
```

Exemplo de Input e Output do Consumidor

- **Input** (Mensagem enviada pelo produtor):
  - Tópico: meu-topico
  - Chave: "chave-exemplo"
  - Valor: "Mensagem de teste"
- **Output** (Saída no console do consumidor):

  Consumido mensagem: chave=chave-exemplo, valor=Mensagem de teste, offset=5

  - **Explicação**: O consumidor lê a mensagem enviada pelo produtor, exibindo a chave (chave-exemplo), o valor (Mensagem de teste), e o offset (5).

Compatibilidade com Java 8, 11 e 17

O código acima é compatível com as versões de Java 8, 11 e 17, pois utilizamos APIs suportadas nas três versões.

Para Java 17, é possível aproveitar os recursos mais modernos, como Pattern Matching e Switch Expressions, se necessário.

Boas Práticas

- **Serialização e Deserialização**: Escolha de serializers apropriados para as mensagens (ex.: Avro, Protobuf) pode otimizar o desempenho.
- **Pooling de Conexões**: Em alta escala, considere configurar pooling de conexões.
- **Assíncrono e Lote**: Para operações de alta escala, preferir envios assíncronos e em lote.

Essa abordagem garante que o Kafka seja utilizado com performance otimizada em diversas versões do Java, atendendo tanto às necessidades de processamento síncrono quanto de paralelismo em sistemas modernos.

Para demonstrar um exemplo de usabilidade real do Kafka em um cenário prático, vou adicionar classes que simulam um fluxo completo de produção e consumo de mensagens em uma aplicação de análise de transações financeiras.

Este exemplo simula uma aplicação onde transações de pagamento são registradas e consumidas por um serviço que processa e analisa esses dados para detecção de fraudes.

Exemplo de Usabilidade: Análise de Transações Financeiras

Neste cenário, teremos duas classes principais:

1. **PaymentTransactionProducer**: Classe que simula o envio de transações para um tópico Kafka.
2. **FraudDetectionConsumer**: Classe que consome essas transações, analisando-as para detectar possíveis fraudes.

Este exemplo de usabilidade foca em como o Kafka pode ser utilizado para transmitir grandes volumes de dados entre serviços de forma eficiente e escalável.

As dependências necessárias são as mesmas mencionadas anteriormente para o Kafka Client, com a adição de qualquer biblioteca que ajude a simular dados para transações, caso seja necessário.

```
<dependency>
   <groupId>org.apache.kafka</groupId>
   <artifactId>kafka-clients</artifactId>
   <version>3.0.0</version>
</dependency>
```

Estrutura de Classes

1. **Modelo da Transação**: Representa uma transação de pagamento.
2. **Produtor Kafka (PaymentTransactionProducer)**: Envia as transações para o Kafka.
3. **Consumidor Kafka (FraudDetectionConsumer)**: Lê e processa as transações para análise de fraudes.

Classe PaymentTransaction

Esta classe representa uma transação de pagamento, com atributos básicos como ID, valor, e status de transação.

```
public class PaymentTransaction {
   private String transactionId;
   private double amount;
   private String status; // Exemplo: "APROVADO", "SUSPEITO"

   public PaymentTransaction(String transactionId, double amount, String status) {
      this.transactionId = transactionId;
```

```
        this.amount = amount;
        this.status = status;
    }

    public String getTransactionId() {
        return transactionId;
    }

    public double getAmount() {
        return amount;
    }

    public String getStatus() {
        return status;
    }

    @Override
    public String toString() {
        return "Transaction ID: " + transactionId + ", Amount: " + amount + ", Status: " + status;
    }
}
```

Classe PaymentTransactionProducer

A classe PaymentTransactionProducer simula o envio de transações para um tópico Kafka chamado transacoes.

```
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerRecord;

import java.util.Properties;

public class PaymentTransactionProducer {

    private static Properties producerProperties() {
        Properties properties = new Properties();
        properties.put("bootstrap.servers", "localhost:9092");
        properties.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
        properties.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");
        return properties;
    }

    public static void main(String[] args) {
        Producer<String, String> producer = new KafkaProducer<>(producerProperties());

        // Simula o envio de várias transações
        for (int i = 0; i < 10; i++) {
            PaymentTransaction transaction = new PaymentTransaction(
                    "transacao-" + i,
                    Math.random() * 1000,
                    i % 2 == 0 ? "APROVADO" : "SUSPEITO"
            );
            producer.send(new ProducerRecord<>("transacoes", transaction.getTransactionId(),
transaction.toString()));
            System.out.printf("Enviado: %s\n", transaction);
        }

        producer.close();
```

```
    }
}
```

Este produtor gera transações de exemplo e envia cada uma para o tópico transacoes.

Ele cria uma nova instância de PaymentTransaction, define alguns valores para simular o status da transação e envia o objeto como uma string para o Kafka.

- **Input**: Valores simulados de transação.
- **Output**: Exibição das transações enviadas para o tópico transacoes.

Classe FraudDetectionConsumer

A classe FraudDetectionConsumer lê transações do tópico transacoes, processa os dados e identifica possíveis fraudes.

```
import org.apache.kafka.clients.consumer.Consumer;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.clients.consumer.ConsumerRecord;

import java.time.Duration;
import java.util.Collections;
import java.util.Properties;

public class FraudDetectionConsumer {

    private static Properties consumerProperties() {
        Properties properties = new Properties();
        properties.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        properties.put(ConsumerConfig.GROUP_ID_CONFIG, "fraud-detection-group");
        properties.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
"org.apache.kafka.common.serialization.StringDeserializer");
        properties.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
"org.apache.kafka.common.serialization.StringDeserializer");
        return properties;
    }

    public static void main(String[] args) {
        Consumer<String, String> consumer = new KafkaConsumer<>(consumerProperties());

        String topic = "transacoes";
        consumer.subscribe(Collections.singletonList(topic));

        try {
            while (true) {
                ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));

                for (ConsumerRecord<String, String> record : records) {
                    String transaction = record.value();

                    // Detecção básica de fraude
                    if (transaction.contains("SUSPEITO")) {
                        System.out.printf("Alerta de fraude! Detalhes da transação: %s\n", transaction);
```

```
                } else {
                    System.out.printf("Transação aprovada: %s\n", transaction);
                }
            }
        }
    } finally {
        consumer.close();
    }
  }
}
```

Este consumidor lê as mensagens do tópico transacoes. Para cada transação recebida, ele verifica o campo status.

Se o status da transação for "SUSPEITO", ele emite um alerta de fraude. Caso contrário, exibe a transação como aprovada.

- **Input**: Mensagens consumidas do tópico transacoes.
- **Output**: Exibição das transações como aprovadas ou suspeitas.

Exemplo de Saída para o Produtor e Consumidor

1. **Produtor** (envio de transações):

   Enviado: Transaction ID: transacao-0, Amount: 547.23, Status: APROVADO
   Enviado: Transaction ID: transacao-1, Amount: 312.54, Status: SUSPEITO
   Enviado: Transaction ID: transacao-2, Amount: 822.11, Status: APROVADO
   ...

2. **Consumidor** (processamento de transações):

   Transação aprovada: Transaction ID: transacao-0, Amount: 547.23, Status: APROVADO
   Alerta de fraude! Detalhes da transação: Transaction ID: transacao-1, Amount: 312.54, Status: SUSPEITO
   Transação aprovada: Transaction ID: transacao-2, Amount: 822.11, Status: APROVADO
   ...

Este exemplo completo mostra como Kafka pode ser usado para implementar um pipeline de dados em um sistema de análise de transações financeiras.

Ele destaca o uso de classes reais para gerar, enviar e consumir dados de forma contínua, com um processamento em tempo real que ajuda na detecção de fraudes.

EducaCiência FastCode para comunidade