**Kafka Consumers: Reading Data from Kafka**

Applications that need to read data from Kafka use a KafkaConsumer to subscribe to Kafka topics and receive messages from these topics. Reading data from Kafka is a bit different than reading data from other messaging systems, and there are a few unique concepts and ideas involved.

**Kafka Consumer Concepts**

To understand how to read data from Kafka, you first need to understand its consumers and consumer groups. The following sections cover those concepts.

**Consumers and Consumer Groups**

If the rate at which producers write messages to the topic exceeds the rate at which your application can validate them? If you are limited to a single consumer reading and processing the data, your application may fall further

and further behind, unable to keep up with the rate of incoming messages. Obviously, there is a need to scale consumption from topics. Just like multiple producers can write to the same topic, we need to allow multiple consumers to read from the same topic, splitting the data among them.

Kafka consumers are typically part of a *consumer group*. When multiple consumers are subscribed to a topic and belong to the same ***consumer group***, each consumer in the group will receive messages from a different subset of the partitions in the topic.

Let’s take topic T1 with four partitions. Now suppose we created a new consumer, C1, which is the only consumer in group G1, and use it to subscribe to topic T1. Consumer C1 will get all messages from all four T1 partitions. See Figure 4-1.



Figure 4-1: One Consumer group with 4 partition

If we add another consumer, C2, to group G1, each consumer will only get messages from two partitions. Perhaps messages from partition 0 and 2 go to C1, and messages from partitions 1 and 3 go to consumer C2. See Figure 4-2.



Figure 4-2: *Four partitions split to two consumers in a group*

If G1 has four consumers, then each will read messages from a single partition. See Figure 4-3.



*Figure 4-3. Four consumers in a group with one partition each*

If we add more consumers to a single group with a single topic than we have partitions, some of the consumers will be idle and get no messages at all. See Figure 4-4.



*Figure 4-4. More consumers in a group than partitions means idle consumers*

The main way we scale data consumption from a Kafka topic is by adding more consumers to a consumer group. It is common for Kafka consumers to do high-latency operations such as write to a database or a time-consuming computation on the data.

In these cases, a single consumer can’t possibly keep up with the rate data flows into a topic, and adding more consumers that share the load by having each consumer own just a subset of the partitions and messages is our main method of scaling. This is a good reason to create topics with a large number of partitions—it allows adding more consumers when the load increases. Keep in mind that there is no point in adding more consumers than you have partitions in a topic—some of the consumers will just be idle.

In the previous example, if we add a new consumer group (G2) with a single consumer, this consumer will get all the messages in topic T1 independent of what G1 is doing. G2 can have more than a single consumer, in which case they will each get a subset of partitions, just like we showed for G1, but G2 as a whole will still get all the messages regardless of other consumer groups. See Figure 4-5.



*Figure 4-5. Adding a new consumer group, both groups receive all messages*

**Consumer Groups and Partition Rebalance**

As we saw in the previous section, consumers in a consumer group share ownership of the partitions in the topics they subscribe to. When we add a new consumer to the group, it starts consuming messages from partitions previously consumed by another consumer. The same thing happens when a consumer shuts down or crashes; it leaves the group, and the partitions it used to consume will be consumed by one of the remaining consumers. Reassignment of partitions to consumers also happens when the topics the consumer group is consuming are modified (e.g., if an administrator adds new partitions).

Moving partition ownership from one consumer to another is called a ***rebalance***. Rebalances are important because they provide the consumer group with high availability and scalability (allowing us to easily and safely add and remove consumers), but in the normal course of events, they can be fairly undesirable. There are two types of rebalances, depending on the partition assignment strategy that the consumer group uses:

1. Eager Rebalances.
2. Cooperative rebalances.

**Creating Kafka Consumer**

The first step to start consuming records is to create a KafkaConsumer instance. Creating a KafkaConsumer is very similar to creating a KafkaProducer—you create a Java Properties instance with the properties you want to pass to the consumer. We will discuss all the properties in depth later in the chapter. To start, we just need to use

the three mandatory properties: **bootstrap.servers, key.deserializer, and value.deserializer**.

The first property, bootstrap.servers, is the connection string to a Kafka cluster. It is used the exact same way as in KafkaProducer (refer to Chapter 3 for details on how this is defined). The other two properties, key.deserializer and value.deserializer, are similar to the serializers defined for the producer, but rather

than specifying classes that turn Java objects to byte arrays, you need to specify classes that can take a byte array and turn it into a Java object.

There is a fourth property, which is not strictly mandatory but is very commonly used. The property is group.id, and it specifies the consumer group the KafkaConsumer instance belongs to. While it is possible to create consumers that do not belong to any consumer group, this is uncommon, so for most of the chapter, we will assume the consumer is part of a group.

The following code snippet shows how to create a KafkaConsumer:

Properties props = new Properties();

props.put("bootstrap.servers", "broker1:9092,broker2:9092");

props.put("group.id", "CountryCounter");

props.put("key.deserializer",

"org.apache.kafka.common.serialization.StringDeserializer");

props.put("value.deserializer",

"org.apache.kafka.common.serialization.StringDeserializer");

KafkaConsumer<String, String> consumer = new KafkaConsumer<String, String>(props);

Subscribing to Topics

Once we create a consumer, the next step is to subscribe to one or more topics. The subscribe() method takes a list of topics as a parameter, so it’s pretty simple to use:

consumer.subscribe(Collections.singletonList("customerCountries"));

Here we simply create a list with a single element: the topic name **customerCountries**.

It is also possible to call subscribe with a regular expression. The expression can match multiple topic names, and if someone creates a new topic with a name that matches, a rebalance will happen almost immediately and the consumers will start consuming from the new topic.

For example, to subscribe to all test topics, we can call:

consumer.subscribe(Pattern.compile("test.\*"));

The Poll Loop:

At the heart of the Consumer API is a simple loop for polling the server for more data. The main body of a consumer will look as follows:

Duration timeout = Duration.ofMillis(100);

while (true) {🡪(1)

ConsumerRecords<String, String> records = consumer.poll(timeout);🡪(2)

for (ConsumerRecord<String, String> record : records) {🡪(3)

System.out.printf("topic = %s, partition = %d, offset = %d, " +

"customer = %s, country = %s\n",

record.topic(), record.partition(), record.offset(),

record.key(), record.value());

int updatedCount = 1;

if (custCountryMap.containsKey(record.value())) {

updatedCount = custCountryMap.get(record.value()) + 1;

}

custCountryMap.put(record.value(), updatedCount);

JSONObject json = new JSONObject(custCountryMap);

System.out.println(json.toString()); 🡪(4)

}

}

1. This is indeed an infinite loop. Consumers are usually long-running applications that continuously poll Kafka for more data. We will show later in the chapter how to cleanly exit the loop and close the consumer.
2. This is the most important line in the chapter. The same way that sharks must keep moving or they die, consumers must keep polling Kafka or they will be considered dead and the partitions they are consuming will be handed to another consumer in the group to continue consuming. The parameter we pass to poll() is a timeout interval and controls how long poll() will block if data is not available in the consumer buffer. If this is set to 0 or if there are records available already, poll() will return immediately; otherwise, it will wait for the specified number of milliseconds.
3. poll() returns a list of records. Each record contains the topic and partition the record came from, the offset of the record within the partition, and, of course, the key and the value of the record. Typically, we want to iterate over the list and process the records individually.
4. Processing usually ends in writing a result in a data store or updating a stored record. Here, the goal is to keep a running count of customers from each country.

Configuring Consumer

1. **fetch.min.bytes**

This property allows a consumer to specify the minimum amount of data that it wants to receive from the broker when fetching records. If a broker receives a request for records from a consumer but the new records amount to fewer bytes than fetch.min.bytes, the broker will wait until more messages are available before sending the records back to the consumer. This reduces the load on both the consumer and the broker as they have to handle fewer back-and-forth messages in cases where the topics don’t have much new activity.

1. **fetch.max.wait.ms**

By setting fetch.min.bytes, you tell Kafka to wait until it has enough data to send before responding to the consumer. fetch.max.wait.ms lets you control how long to wait. By default, Kafka will wait up to 500 ms. This results in up to 500 ms of extra latency in case there is not enough data flowing to the Kafka topic to satisfy the minimum amount of data to return. If you want to limit the potential latency (usually due to SLAs controlling the maximum latency of the application), you can set fetch.max.wait.ms to a lower value.

1. **max.partion.fetch.bytes**

This property controls the maximum number of bytes the server will return per partition. The default is 1 MB, which means that when KafkaConsumer.poll() returns ConsumerRecords, the record object will use at most max.partition.fetch.bytes per partition assigned to the consumer. So if a topic has 20 partitions, and you have 5 consumers, each consumer will need to have 4 MB of memory available for ConsumerRecords.

1. **session.timeout.ms**

The amount of time a consumer can be out of contact with the brokers while still considered alive defaults to 10 seconds. If more than session.timeout.ms passes without the consumer sending a heartbeat to the group coordinator, it is considered dead and the group coordinator will trigger a rebalance of the consumer group to allocate partitions from the dead consumer to the other consumers in the group.

1. **Enable.auto.commit**

We’ll discuss the different options for committing offsets later in this chapter. This parameter controls whether the consumer will commit offsets automatically, and defaults to true. Set it to false if you prefer to control when offsets are committed, which is necessary to minimize duplicates and avoid missing data. If you set enable.auto.commit to true, then you might also want to control how frequently offsets will be committed using auto.commit.interval.ms.

1. **partition.assignment.strategy**

We learned that partitions are assigned to consumers in a consumer group. A PartitionAssignor is a class that, given consumers and topics they subscribed to, decides which partitions will be assigned to which consumer. By default, Kafka has two assignment strategies:

* 1. Range
  2. RoundRobin

1. **client.id**

This can be any string, and will be used by the brokers to identify messages sent from the client. It is used in logging and metrics, and for quotas.

1. **max.poll.records**

This controls the maximum number of records that a single call to poll() will return. This is useful to help control the amount of data your application will need to process in the polling loop.