EBook Gratis

APRENDIZAJE apache-kafka

Free unaffiliated eBook created from
Stack Overflow contributors.

#apache-kafka

Tabla de contenido

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: apache-kafka

It is an unofficial and free apache-kafka ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official apache-kafka.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con apache-kafka

Observaciones

Kafka es un sistema de mensajería de publicación-suscripción de alto rendimiento implementado como un servicio de registro de confirmación duplicado, particionado y distribuido.

Tomado del sitio oficial de Kafka

> **Rápido**
>
> Un solo agente de Kafka puede manejar cientos de megabytes de lecturas y escrituras por segundo de miles de clientes.
>
> **Escalable**
>
> Kafka está diseñado para permitir que un solo clúster sirva como la red troncal de datos central para una organización grande. Puede expandirse de forma elástica y transparente sin tiempo de inactividad. Los flujos de datos se dividen en particiones y se distribuyen en un grupo de máquinas para permitir flujos de datos mayores que la capacidad de cualquier máquina individual y para permitir grupos de consumidores coordinados
>
> **Durable**
>
> Los mensajes se guardan en el disco y se replican dentro del clúster para evitar la pérdida de datos. Cada agente puede manejar terabytes de mensajes sin impacto en el rendimiento.
>
> **Distribuido por Diseño**
>
> Kafka tiene un diseño moderno centrado en el clúster que ofrece durabilidad sólida y garantías de tolerancia a fallas.

Examples

Instalación o configuración

Paso 1 Instalar Java 7 u 8

Paso 2 Descargue Apache Kafka en: http://kafka.apache.org/downloads.html

Por ejemplo, intentaremos descargar Apache Kafka 0.10.0.0

Paso 3 Extrae el archivo comprimido.

En Linux:

```
tar -xzf kafka_2.11-0.10.0.0.tgz
```

En la ventana: Haga clic derecho -> Extraer aquí

Paso 4 . Iniciar zookeeper

```
cd kafka_2.11-0.10.0.0
```

Linux:

```
bin/zookeeper-server-start.sh config/zookeeper.properties
```

Windows:

```
bin/windows/zookeeper-server-start.bat config/zookeeper.properties
```

Paso 5 Iniciar el servidor Kafka

Linux:

```
bin/kafka-server-start.sh config/server.properties
```

Windows:

```
bin/windows/kafka-server-start.bat config/server.properties
```

Introducción

Apache Kafka ™ es una plataforma de transmisión distribuida.

Lo que significa

1-Te permite publicar y suscribirte a flujos de registros. En este sentido, es similar a una cola de mensajes o un sistema de mensajería empresarial.

2-Le permite almacenar flujos de registros de forma tolerante a fallos.

3-Te permite procesar flujos de registros a medida que ocurren.

Se utiliza para dos amplias clases de aplicación:

1-Creación de flujos de datos en tiempo real para obtener datos de manera confiable entre sistemas o aplicaciones

2-Creación de aplicaciones de transmisión en tiempo real que transforman o reaccionan a los flujos de datos

> Los scripts de la consola Kafka son diferentes para las plataformas basadas en Unix y Windows. En los ejemplos, es posible que necesite agregar la extensión de acuerdo con su plataforma. Linux: scripts ubicados en `bin/` con extensión `.sh` . Windows: scripts ubicados en `bin\windows\` y con extensión `.bat` .

Instalación

Paso 1: Descarga el código y descomprímelo:

```
tar -xzf kafka_2.11-0.10.1.0.tgz
cd kafka_2.11-0.10.1.0
```

Paso 2: inicia el servidor.

> para poder eliminar temas más adelante, abra `server.properties` y establezca `delete.topic.enable` en true.

Kafka depende en gran medida del cuidador del zoológico, por lo que primero debes comenzar. Si no lo tiene instalado, puede usar el script de conveniencia empaquetado con kafka para obtener una instancia de ZooKeeper de un solo nodo rápida y sucia.

```
zookeeper-server-start config/zookeeper.properties
kafka-server-start config/server.properties
```

Paso 3: asegúrate de que todo está funcionando

Ahora debería tener un guardián del zoológico escuchando a `localhost:2181` y un solo corredor de kafka en `localhost:6667` .

Crear un tema

Solo tenemos un agente, por lo que creamos un tema sin factor de replicación y solo una partición:

```
kafka-topics --zookeeper localhost:2181 \
    --create \
    --replication-factor 1 \
    --partitions 1 \
    --topic test-topic
```

Revisa tu tema:

```
kafka-topics --zookeeper localhost:2181 --list
test-topic

kafka-topics --zookeeper localhost:2181 --describe --topic test-topic
Topic:test-topic    PartitionCount:1    ReplicationFactor:1 Configs:
Topic: test-topic   Partition: 0    Leader: 0   Replicas: 0 Isr: 0
```

enviar y recibir mensajes

Lanzar un consumidor:

```
kafka-console-consumer --bootstrap-server localhost:9092 --topic test-topic
```

En otra terminal, lanza un productor y envía algunos mensajes. De forma predeterminada, la herramienta envía cada línea como un mensaje separado al intermediario, sin codificación especial. Escriba algunas líneas y salga con CTRL + D o CTRL + C:

```
kafka-console-producer --broker-list localhost:9092 --topic test-topic
a message
another message
^D
```

Los mensajes deben aparecer en el terminal del consumidor.

Deja de kafka

```
kafka-server-stop
```

iniciar un cluster multi-broker

Los ejemplos anteriores utilizan un solo corredor. Para configurar un cluster real, solo necesitamos iniciar más de un servidor kafka. Ellos se coordinarán automáticamente.

Paso 1: para evitar la colisión, creamos un archivo `server.properties` para cada agente y cambiamos las propiedades de configuración de `id` , `port` y `logfile` .

Dupdo:

```
cp config/server.properties config/server-1.properties
cp config/server.properties config/server-2.properties
```

Editar propiedades para cada archivo, por ejemplo:

```
vim config/server-1.properties
broker.id=1
listeners=PLAINTEXT://:9093
log.dirs=/usr/local/var/lib/kafka-logs-1

vim config/server-2.properties
broker.id=2
listeners=PLAINTEXT://:9094
log.dirs=/usr/local/var/lib/kafka-logs-2
```

Paso 2: iniciar los tres corredores:

```
    kafka-server-start config/server.properties &
    kafka-server-start config/server-1.properties &
    kafka-server-start config/server-2.properties &
```

Crear un tema replicado

```
kafka-topics --zookeeper localhost:2181 --create --replication-factor 3 --partitions 1 --topic
replicated-topic

kafka-topics --zookeeper localhost:2181 --describe --topic replicated-topic
Topic:replicated-topic  PartitionCount:1    ReplicationFactor:3 Configs:
Topic: replicated-topic Partition: 0    Leader: 1   Replicas: 1,2,0 Isr: 1,2,0
```

Esta vez, hay más información:

- "líder" es el nodo responsable de todas las lecturas y escrituras de la partición dada. Cada nodo será el líder de una parte seleccionada aleatoriamente de las particiones.
- "réplicas" es la lista de nodos que replican el registro para esta partición, independientemente de si son los líderes o incluso si están actualmente activos.
- "isr" es el conjunto de réplicas "in-sync". Este es el subconjunto de la lista de réplicas que actualmente está viva y está al día con el líder.

Tenga en cuenta que el tema creado anteriormente se mantiene sin cambios.

prueba de tolerancia a fallos

Publicar algún mensaje al nuevo tema:

```
kafka-console-producer --broker-list localhost:9092 --topic replicated-topic
hello 1
hello 2
^C
```

Mata al líder (1 en nuestro ejemplo). En Linux:

```
ps aux | grep server-1.properties
kill -9 <PID>
```

En Windows:

```
wmic process get processid,caption,commandline | find "java.exe" | find "server-1.properties"
taskkill /pid <PID> /f
```

Mira lo que pasó:

```
kafka-topics --zookeeper localhost:2181  --describe --topic replicated-topic
Topic:replicated-topic  PartitionCount:1    ReplicationFactor:3 Configs:
Topic: replicated-topic Partition: 0    Leader: 2   Replicas: 1,2,0 Isr: 2,0
```

El liderazgo ha cambiado al corredor 2 y "1" en no sincronizado más. Pero los mensajes todavía están allí (usa al consumidor para verificar por ti mismo).

Limpiar

Borre los dos temas usando:

```
kafka-topics --zookeeper localhost:2181 --delete --topic test-topic
kafka-topics --zookeeper localhost:2181 --delete --topic replicated-topic
```

Lea Empezando con apache-kafka en línea: https://riptutorial.com/es/apache-kafka/topic/1986/empezando-con-apache-kafka

Capítulo 2: Grupos de consumidores y gestión de compensaciones

Parámetros

Parámetro	Descripción
Identificación del grupo	El nombre del Grupo de Consumidores.
enable.auto.commit	Confirmar automáticamente las compensaciones; *por defecto: verdadero*
auto.commit.interval.ms	El retraso mínimo en milisegundos entre a las confirmaciones (requiere `enable.auto.commit=true`); *por defecto: 5000* .
auto.offset.reset	Qué hacer cuando no se encuentra un desplazamiento confirmado válido; *por defecto: la última* . (+)
(+) Valores posibles	**Descripción**
más temprano	Restablecer automáticamente el desplazamiento al primer desplazamiento.
último	Restablecer automáticamente el desplazamiento a la última compensación.
ninguna	Lanzar excepción al consumidor si no se encuentra un desplazamiento anterior para el grupo de consumidores.
Algo más	Lanzar excepción al consumidor.

Examples

¿Qué es un grupo de consumidores?

A partir de Kafka 0.9, el nuevo cliente de alto nivel KafkaConsumer está disponible. Explota un nuevo protocolo Kafka incorporado que permite combinar múltiples consumidores en un llamado *Grupo de Consumidores* . Un grupo de consumidores puede describirse como un único consumidor lógico que se suscribe a un conjunto de temas. Las partes de todos los temas se asignan a los consumidores físicos dentro del grupo, de modo que cada una de ellas se asigna a un solo consumidor (un solo consumidor puede tener varias particiones asignadas). Los consumidores individuales que pertenecen al mismo grupo pueden ejecutar en diferentes hosts de manera distribuida.

Los grupos de consumidores se identifican a través de su `group.id` . Para hacer que una instancia de cliente específica sea miembro de un grupo de consumidores, es suficiente asignar los grupos `group.id` a este cliente, a través de la configuración del cliente:

```
Properties props = new Properties();
props.put("group.id", "groupName");
// ...some more properties required
new KafkaConsumer<K, V>(config);
```

Por lo tanto, todos los consumidores que se conectan al mismo clúster Kafka y usan el mismo `group.id` forman un grupo de consumidores. Los consumidores pueden dejar un grupo en cualquier momento y los nuevos consumidores pueden unirse a un grupo en cualquier momento. En ambos casos, se activa un llamado *rebalanceo* y las particiones se reasignan con el Grupo de consumidores para garantizar que cada partición sea procesada por un consumidor dentro del grupo.

Preste atención, que incluso un solo `KafkaConsumer` forma un Grupo de Consumidores consigo mismo como miembro único.

Gestión de la compensación del consumidor y tolerancia a fallos

KafkaConsumers solicita mensajes a un agente de Kafka a través de una llamada a `poll()` y su progreso se rastrea a través de *compensaciones* . Cada mensaje dentro de cada partición de cada tema tiene un llamado desplazamiento asignado: su número de secuencia lógica dentro de la partición. Un `KafkaConsumer` rastrea su compensación actual para cada partición que se le asigna. Preste atención, que los corredores de Kafka no están al tanto de las compensaciones actuales de los consumidores. Por lo tanto, en la `poll()` el consumidor debe enviar sus compensaciones actuales al intermediario, de modo que el intermediario pueda devolver los mensajes correspondientes, es decir ,. Mensajes con mayor desplazamiento consecutivo. Por ejemplo, supongamos que tenemos un solo tema de partición y un solo consumidor con la compensación actual 5. En la `poll()` el consumidor envía una compensación al agente y los mensajes de devolución del agente para las compensaciones 6,7,8, ...

Debido a que los consumidores rastrean sus propias compensaciones, esta información podría perderse si un consumidor falla. Por lo tanto, las compensaciones deben almacenarse de manera confiable, de modo que al reiniciar, un consumidor puede recoger su compensación anterior y volver a clasificarla donde la dejó. En Kafka, hay soporte incorporado para esto a través de *confirmaciones de compensación* . El nuevo `KafkaConsumer` puede comprometer su compensación actual a Kafka y Kafka almacena esas compensaciones en un tema especial llamado `__consumer_offsets` . El almacenamiento de las compensaciones dentro de un tema Kafka no solo es tolerante a fallos, sino que también permite reasignar particiones a otros consumidores durante un reequilibrio. Debido a que todos los consumidores de un Grupo de consumidores pueden acceder a todas las compensaciones confirmadas de todas las particiones, en el rebalanceo, un consumidor que obtiene una nueva partición asignada simplemente lee el desplazamiento confirmado de esta partición del tema `__consumer_offsets` y reanuda el lugar donde quedó el antiguo consumidor.

Cómo cometer compensaciones

KafkaConsumers puede asignar compensaciones automáticamente en segundo plano (parámetro de configuración `enable.auto.commit = true`), cuál es la configuración predeterminada. Estas confirmaciones automáticas se realizan dentro de `poll()` (que normalmente se llama en un bucle). La frecuencia con la que se deben confirmar las compensaciones, se puede configurar a través de `auto.commit.interval.ms` . Debido a que las confirmaciones automáticas están integradas en `poll()` y el código de usuario llama a `poll()` , este parámetro define un límite inferior para el intervalo entre confirmaciones.

Como alternativa a la confirmación automática, las compensaciones también se pueden gestionar manualmente. Para esto, la confirmación automática debe estar deshabilitada (`enable.auto.commit = false`). Para la `KafkaConsumers` manual, `KafkaConsumers` ofrece dos métodos, a saber, commitSync () y commitAsync () . Como su nombre lo indica, `commitSync()` es una llamada de bloqueo, que se devuelve después de que las compensaciones se confirmaron correctamente, mientras que `commitAsync()` devuelve inmediatamente. Si desea saber si una confirmación fue exitosa o no, puede proporcionar un controlador de devolución de llamada (`OffsetCommitCallback`) un parámetro de método. Preste atención, que en ambas llamadas de confirmación, el consumidor realiza las compensaciones de la última llamada a `poll()` . Por ejemplo. Asumamos un solo tema de partición con un solo consumidor y la última llamada a `poll()` devuelve mensajes con compensaciones 4,5,6. En la confirmación, la compensación 6 se confirmará porque esta es la última compensación seguida por el cliente consumidor. Al mismo tiempo, tanto `commitSync()` como `commitAsync()` permiten tener más control sobre qué compensación desea comprometer: si usa las sobrecargas correspondientes que le permiten especificar un `Map<TopicPartition, OffsetAndMetadata>` el consumidor solo confirmará las compensaciones especificadas (es decir, el mapa puede contener cualquier subconjunto de particiones asignadas, y el desplazamiento especificado puede tener cualquier valor).

Semántica de compensaciones comprometidas.

Un desplazamiento confirmado indica que todos los mensajes hasta este desplazamiento ya se han procesado. Por lo tanto, como las compensaciones son números consecutivos, la compensación de `X` compromete implícitamente todas las compensaciones más pequeñas que `X` Por lo tanto, no es necesario comprometer cada desplazamiento de forma individual y, al mismo tiempo, se cometen varias compensaciones a la vez, solo se realiza la compensación más grande.

Tenga en cuenta que, por diseño, también es posible comprometer una compensación menor que la última compensación confirmada. Esto se puede hacer, si los mensajes deben leerse por segunda vez.

Procesamiento de garantías

El uso de la confirmación automática proporciona al menos una vez la semántica de procesamiento. El supuesto subyacente es que solo se llama a `poll()` después de que todos los mensajes entregados previamente se procesaron correctamente. Esto garantiza que no se pierda

ningún mensaje porque se produce una confirmación *después del* procesamiento. Si un consumidor falla antes de una confirmación, todos los mensajes después de la última confirmación se reciben de Kafka y se procesan nuevamente. Sin embargo, este reintento puede dar como resultado duplicados, ya que algunos mensajes de la última llamada a `poll()` pueden haberse procesado, pero el error ocurrió justo antes de la llamada de confirmación automática.

Si se requiere semántica de procesamiento a lo sumo una vez, se debe deshabilitar la confirmación automática y se debe realizar un `commitSync()` manual directamente después de la `poll()` . Después, los mensajes se procesan. Esto garantiza que los mensajes se confirmen *antes de* que se procesen y, por lo tanto, nunca se lean una segunda vez. Por supuesto, algún mensaje podría perderse en caso de fallo.

¿Cómo puedo leer el tema desde su principio?

Existen múltiples estrategias para leer un tema desde su inicio. Para explicarlos, primero debemos entender qué sucede en el inicio del consumidor. Al inicio de un consumidor, sucede lo siguiente:

1. unirse al grupo de consumidores configurado, que activa un rebalanceo y asigna particiones al consumidor
2. buscar compensaciones comprometidas (para todas las particiones que se asignaron al consumidor)
3. para todas las particiones con compensación válida, reanudar desde esta compensación
4. para todas las particiones con compensación no válida, configure la compensación de inicio de acuerdo con el parámetro de configuración `auto.offset.reset`

Iniciar un nuevo grupo de consumidores

Si desea procesar un tema desde su inicio, puede iniciar un nuevo grupo de consumidores (es decir, elegir un `group.id` no `group.id`) y establecer `auto.offset.reset = earliest` . Debido a que no hay compensaciones confirmadas para un nuevo grupo, el restablecimiento de la compensación automática se activará y el tema se consumirá desde el principio. Preste atención, que en el reinicio del consumidor, si usa el mismo `group.id` nuevamente, no volverá a leer el tema desde el principio, sino que continuará donde lo dejó. Por lo tanto, para esta estrategia, deberá asignar un nuevo `group.id` cada vez que desee leer un tema desde el principio.

Reutilizar el mismo ID de grupo

Para evitar configurar un nuevo `group.id` cada vez que quiera leer un tema desde su inicio, puede desactivar el compromiso automático (a través de `enable.auto.commit = false`) antes de iniciar el consumidor por primera vez (utilizando un `group.id` no utilizado) `group.id` y configuración `auto.offset.reset = earliest`). Además, no debe realizar ninguna compensación manualmente. Debido a que las compensaciones nunca se comprometen con esta estrategia, al reiniciar, el consumidor leerá el tema desde el principio nuevamente.

Sin embargo, esta estrategia tiene dos desventajas:

1. no es tolerante a fallas
2. reequilibrio de grupo no funciona como se esperaba

(1) Debido a que las compensaciones nunca se comprometen, un consumidor fallido y detenido se manejan de la misma manera en el reinicio. Para ambos casos, el tema será consumido desde su inicio. (2) Debido a que los desplazamientos nunca se comprometen, al rebalancear las particiones recién asignadas serán consumidores desde el principio.

Por lo tanto, esta estrategia solo funciona para grupos de consumidores con un solo consumidor y solo debe utilizarse para fines de desarrollo.

Reutilice la misma ID de grupo y confirme

Si desea ser tolerante a fallos y / o utilizar varios consumidores en su Grupo de consumidores, es obligatorio realizar compensaciones. Por lo tanto, si desea leer un tema desde el principio, debe manipular las compensaciones confirmadas en el inicio del consumidor. Para esto, `KafkaConsumer` proporciona tres métodos `seek()` , `seekToBeginning()` y `seekToEnd()` . Mientras que `seek()` se puede usar para establecer un desplazamiento arbitrario, el segundo y tercer método se pueden usar para buscar el principio o el final de una partición, respectivamente. Por lo tanto, en caso de fallo y en la búsqueda de reinicio del consumidor se omitiría y el consumidor puede reanudar donde lo dejó. Para consumidor-stop-and-restart-from- `seekToBeginning()` , se `seekToBeginning()` explícitamente antes de ingresar a su bucle de `poll()` . Tenga en cuenta que `seekXXX()` solo se puede usar después de que un consumidor se unió a un grupo; por lo tanto, se requiere que realice una "encuesta ficticia" antes de usar `seekXXX()` . El código general sería algo como esto:

```
if (consumer-stop-and-restart-from-beginning) {
    consumer.poll(0); // dummy poll() to join consumer group
    consumer.seekToBeginning(...);
}

// now you can start your poll() loop
while (isRunning) {
    for (ConsumerRecord record : consumer.poll(0)) {
        // process a record
    }
}
```

Lea Grupos de consumidores y gestión de compensaciones en línea: https://riptutorial.com/es/apache-kafka/topic/5449/grupos-de-consumidores-y-gestion-de-compensaciones

Capítulo 3: herramientas de consola kafka

Introducción

Kafka ofrece herramientas de línea de comandos para administrar temas, grupos de consumidores, consumir y publicar mensajes, etc.

Importante : los scripts de la consola Kafka son diferentes para las plataformas basadas en Unix y Windows. En los ejemplos, es posible que necesite agregar la extensión de acuerdo con su plataforma.

Linux : scripts ubicados en `bin/` con extensión `.sh` .

Windows : scripts ubicados en `bin\windows\` y con extensión `.bat` .

Examples

kafka-temas

Esta herramienta le permite enumerar, crear, alterar y describir temas.

Lista de temas:

```
kafka-topics  --zookeeper localhost:2181 --list
```

Crear un tema:

```
kafka-topics  --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --
topic test
```

crea un tema con una partición y sin replicación.

Describe un tema:

```
kafka-topics  --zookeeper localhost:2181 --describe --topic test
```

Alterar un tema:

```
# change configuration
kafka-topics  --zookeeper localhost:2181 --alter --topic test --config
max.message.bytes=128000
# add a partition
kafka-topics  --zookeeper localhost:2181 --alter --topic test --partitions 2
```

(Cuidado: Kafka no admite la reducción del número de particiones de un tema) (consulte esta lista de propiedades de configuración)

productor de consola kafka

Esta herramienta te permite producir mensajes desde la línea de comandos.

Enviar mensajes simples de cadena a un tema:

```
kafka-console-producer --broker-list localhost:9092 --topic test
here is a message
here is another message
^D
```

(cada nueva línea es un mensaje nuevo, escriba ctrl + D o ctrl + C para detener)

Enviar mensajes con claves:

```
kafka-console-producer --broker-list localhost:9092 --topic test-topic \
        --property parse.key=true \
        --property key.separator=,
key 1, message 1
key 2, message 2
null, message 3
^D
```

Enviar mensajes desde un archivo:

```
kafka-console-producer --broker-list localhost:9092 --topic test_topic < file.log
```

kafka-console-consumer

Esta herramienta te permite consumir mensajes de un tema.

> para usar la implementación anterior del consumidor, reemplace `--bootstrap-server` con `--zookeeper` .

Mostrar mensajes simples:

```
kafka-console-consumer --bootstrap-server localhost:9092 --topic test
```

Consumir mensajes antiguos:

Para ver los mensajes más antiguos, puede usar la opción `--from-beginning` .

Mostrar mensajes de valor-clave :

```
kafka-console-consumer  --bootstrap-server localhost:9092 --topic test-topic \
    --property print.key=true \
    --property key.separator=,
```

kafka-simple-consumidor-shell

Este consumidor es una herramienta de bajo nivel que le permite consumir mensajes de particiones, compensaciones y réplicas específicas.

Parámetros útiles:

- `parition` : la partición específica para consumir a partir de (por defecto a todos)
- `offset` : el offset inicial. Usa `-2` para consumir mensajes desde el principio, `-1` para consumir desde el final.
- `max-messages` : número de mensajes para imprimir
- `replica` : la réplica, predeterminada para el broker-leader (-1)

Ejemplo:

```
kafka-simple-consumer-shell  \
    --broker-list localhost:9092 \
    --partition 1 \
    --offset 4 \
    --max-messages 3 \
    --topic test-topic
```

muestra 3 mensajes de la partición 1 que comienzan en el desplazamiento 4 del tema prueba-tema.

kafka-grupos de consumidores

Esta herramienta le permite enumerar, describir o eliminar grupos de consumidores. Eche un vistazo a este artículo para obtener más información sobre los grupos de consumidores.

> si aún usa la implementación anterior del consumidor, reemplace `--bootstrap-server` con `--zookeeper` .

Lista de grupos de consumidores:

```
kafka-consumer-groups  --bootstrap-server localhost:9092 --list
octopus
```

Describa un grupo de consumidores:

```
kafka-consumer-groups  --bootstrap-server localhost:9092 --describe --group octopus
GROUP          TOPIC           PARTITION  CURRENT-OFFSET  LOG-END-OFFSET  LAG       OWNER
octopus        test-topic      0          15              15              0         octopus-
1/127.0.0.1
octopus        test-topic      1          14              15              1         octopus-
2_/127.0.0.1
```

Observaciones : en la salida anterior,

- `current-offset` es el último offset comprometido de la instancia del consumidor,
- `log-end-offset` es el desplazamiento más alto de la partición (por lo tanto, sumar esta columna le da el número total de mensajes para el tema)
- `lag` es la diferencia entre la compensación del consumidor actual y la compensación más

alta, por lo tanto, cuán lejos está el consumidor,
- `owner` es el `client.id` del consumidor (si no se especifica, se muestra uno predeterminado).

Eliminar un grupo de consumidores:

> la eliminación solo está disponible cuando los metadatos del grupo se almacenan en zookeeper (antigua api del consumidor). Con la nueva API para el consumidor, el agente maneja todo lo que incluye la eliminación de metadatos: el grupo se elimina automáticamente cuando expira el último desplazamiento confirmado para el grupo.

```
kafka-consumer-groups --bootstrap-server localhost:9092 --delete --group octopus
```

Lea herramientas de consola kafka en línea: https://riptutorial.com/es/apache-kafka/topic/8990/herramientas-de-consola-kafka

Capítulo 4: Productor / Consumidor en Java

Introducción

Este tema muestra cómo producir y consumir registros en Java.

Examples

SimpleConsumer (Kafka> = 0.9.0)

La versión 0.9 de Kafka introdujo un rediseño completo del consumidor kafka. Si está interesado en el antiguo `SimpleConsumer` (0.8.X), eche un vistazo a esta página . Si su instalación de Kafka es más reciente que 0.8.X, los siguientes códigos deberían funcionar de manera inmediata.

Configuración e inicialización

> Kafka 0.9 ya no es compatible con Java 6 o Scala 2.9. Si aún está en Java 6, considere actualizar a una versión compatible.

Primero, cree un proyecto de Maven y agregue la siguiente dependencia en su pom:

```
<dependencies>
    <dependency>
        <groupId>org.apache.kafka</groupId>
        <artifactId>kafka-clients</artifactId>
        <version>0.9.0.1</version>
    </dependency>
</dependencies>
```

Nota : no olvide actualizar el campo de versión para las últimas versiones (ahora> 0.10).

El consumidor se inicializa utilizando un objeto `Properties` . Hay muchas propiedades que le permiten ajustar el comportamiento del consumidor. A continuación se muestra la configuración mínima necesaria:

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("group.id", "consumer-tutorial");
props.put("key.deserializer", StringDeserializer.class.getName());
props.put("value.deserializer", StringDeserializer.class.getName());
```

Los `bootstrap-servers` son una lista inicial de agentes para que el consumidor pueda descubrir el resto del clúster. No es necesario que sean todos los servidores del clúster: el cliente determinará el conjunto completo de intermediarios activos de los intermediarios en esta lista.

El `deserializer` le dice al consumidor cómo interpretar / deserializar las claves y valores del

mensaje. Aquí, usamos el `StringDeserializer` .

Finalmente, el `group.id` corresponde al grupo de consumidores de este cliente. Recuerde: todos los consumidores de un grupo de consumidores dividirán los mensajes entre ellos (kafka actuando como una cola de mensajes), mientras que los consumidores de diferentes grupos de consumidores recibirán los mismos mensajes (kafka actuando como un sistema de publicación-suscripción).

Otras propiedades útiles son:

- `auto.offset.reset` : controla qué hacer si el desplazamiento almacenado en Zookeeper falta o está fuera de rango. Los valores posibles son los `latest` y los `earliest` . Cualquier otra cosa lanzará una excepción;

- `enable.auto.commit` : si es `true` (predeterminado), el desplazamiento del consumidor se `auto.commit.interval.ms` periódicamente (ver `auto.commit.interval.ms`) guardado en el fondo. Establecerlo en `false` y usar `auto.offset.reset=earliest` - es para determinar desde dónde debe comenzar el consumidor en caso de que no se encuentre información de compensación confirmada. `earliest` medios desde el inicio de la partición del tema asignado. `latest` medios del mayor número de compensaciones comprometidas disponibles para la partición. Sin embargo, el consumidor de Kafka siempre se reanudará desde el último desplazamiento confirmado siempre que se encuentre un registro de desplazamiento válido (es decir, ignorando `auto.offset.reset` . El mejor ejemplo es cuando un grupo de consumidores nuevo se suscribe a un tema. Esto es cuando se usa `auto.offset.reset` para determinar si comenzar desde el principio (más temprano) o el final (más reciente) del tema.

- `session.timeout.ms` : un tiempo de espera de sesión asegura que el bloqueo se liberará si el consumidor se bloquea o si una partición de red aísla al consumidor del coordinador. En efecto:

  > Cuando forman parte de un grupo de consumidores, a cada consumidor se le asigna un subconjunto de las particiones de los temas a los que se ha suscrito. Esto es básicamente un bloqueo de grupo en esas particiones. Mientras se mantenga el bloqueo, ningún otro miembro del grupo podrá leer de ellos. Cuando su consumidor está sano, esto es exactamente lo que quiere. Es la única forma de evitar el consumo duplicado. Pero si el consumidor muere debido a una falla de la máquina o la aplicación, necesita que se libere ese bloqueo para que las particiones puedan asignarse a un miembro sano. fuente

La lista completa de propiedades está disponible aquí http://kafka.apache.org/090/documentation.html#newconsumerconfigs .

Creación del consumidor y suscripción al tema.

Una vez que tenemos las propiedades, crear un consumidor es fácil:

```
KafkaConsumer<String, String> consumer = new KafkaConsumer<>( props );
consumer.subscribe( Collections.singletonList( "topic-example" ) );
```

Una vez que se haya suscrito, el consumidor puede coordinar con el resto del grupo para obtener su asignación de partición. Todo esto se maneja automáticamente cuando empiezas a consumir datos.

Encuesta basica

El consumidor debe poder obtener datos en paralelo, potencialmente de muchas particiones para muchos temas que probablemente se distribuyen entre muchos corredores. Afortunadamente, todo esto se maneja automáticamente cuando comienza a consumir datos. Para hacer eso, todo lo que necesita hacer es llamar a la `poll` en un bucle y el consumidor se encarga del resto.

`poll` devuelve un conjunto (posiblemente vacío) de mensajes de las particiones que fueron asignadas.

```
while( true ){
    ConsumerRecords<String, String> records = consumer.poll( 100 );
    if( !records.isEmpty() ){
        StreamSupport.stream( records.spliterator(), false ).forEach( System.out::println );
    }
}
```

El código

Ejemplo basico

Este es el código más básico que puede usar para obtener mensajes de un tema kafka.

```
public class ConsumerExample09{

    public static void main( String[] args ){

        Properties props = new Properties();
        props.put( "bootstrap.servers", "localhost:9092" );
        props.put( "key.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer" );
        props.put( "value.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer" );
        props.put( "auto.offset.reset", "earliest" );
        props.put( "enable.auto.commit", "false" );
        props.put( "group.id", "octopus" );

        try( KafkaConsumer<String, String> consumer = new KafkaConsumer<>( props ) ){
            consumer.subscribe( Collections.singletonList( "test-topic" ) );

            while( true ){
                // poll with a 100 ms timeout
                ConsumerRecords<String, String> records = consumer.poll( 100 );
```

```
                if( records.isEmpty() ) continue;
                StreamSupport.stream( records.spliterator(), false ).forEach(
System.out::println );
            }
        }
    }
}
```

Ejemplo ejecutable

> El consumidor está diseñado para ejecutarse en su propio hilo. No es seguro para uso multiproceso sin sincronización externa y probablemente no sea una buena idea intentarlo.

A continuación se muestra una tarea sencilla de Ejecutar que inicializa al consumidor, se suscribe a una lista de temas y ejecuta el bucle de sondeo indefinidamente hasta que se apaga externamente.

```
public class ConsumerLoop implements Runnable{
    private final KafkaConsumer<String, String> consumer;
    private final List<String> topics;
    private final int id;

    public ConsumerLoop( int id, String groupId, List<String> topics ){
        this.id = id;
        this.topics = topics;
        Properties props = new Properties();
        props.put( "bootstrap.servers", "localhost:9092");
        props.put( "group.id", groupId );
        props.put( "auto.offset.reset", "earliest" );
        props.put( "key.deserializer", StringDeserializer.class.getName() );
        props.put( "value.deserializer", StringDeserializer.class.getName() );
        this.consumer = new KafkaConsumer<>( props );
    }

    @Override
    public void run(){
        try{
            consumer.subscribe( topics );

            while( true ){
                ConsumerRecords<String, String> records = consumer.poll( Long.MAX_VALUE );
                StreamSupport.stream( records.spliterator(), false ).forEach(
System.out::println );
            }
        }catch( WakeupException e ){
            // ignore for shutdown
        }finally{
            consumer.close();
        }
    }


    public void shutdown(){
        consumer.wakeup();
    }
}
```

Tenga en cuenta que usamos un tiempo de espera de `Long.MAX_VALUE` durante la encuesta, por lo que esperará indefinidamente un mensaje nuevo. Para cerrar correctamente el consumidor, es importante llamar a su método `shutdown()` antes de finalizar la aplicación.

Un conductor podría usarlo así:

```
public static void main( String[] args ){

    int numConsumers = 3;
    String groupId = "octopus";
    List<String> topics = Arrays.asList( "test-topic" );

    ExecutorService executor = Executors.newFixedThreadPool( numConsumers );
    final List<ConsumerLoop> consumers = new ArrayList<>();

    for( int i = 0; i < numConsumers; i++ ){
        ConsumerLoop consumer = new ConsumerLoop( i, groupId, topics );
        consumers.add( consumer );
        executor.submit( consumer );
    }

    Runtime.getRuntime().addShutdownHook( new Thread(){
        @Override
        public void run(){
            for( ConsumerLoop consumer : consumers ){
                consumer.shutdown();
            }
            executor.shutdown();
            try{
                executor.awaitTermination( 5000, TimeUnit.MILLISECONDS );
            }catch( InterruptedException e ){
                e.printStackTrace();
            }
        }
    } );
}
```

SimpleProducer (kafka> = 0.9)

Configuración e inicialización

Primero, cree un proyecto de Maven y agregue la siguiente dependencia en su pom:

```
 <dependencies>
     <dependency>
         <groupId>org.apache.kafka</groupId>
         <artifactId>kafka-clients</artifactId>
         <version>0.9.0.1</version>
     </dependency>
 </dependencies>
```

El productor se inicializa utilizando un objeto `Properties` . Hay muchas propiedades que le permiten ajustar el comportamiento del productor. A continuación se muestra la configuración mínima necesaria:

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");
props.put("client.id", "simple-producer-XX");
```

Los `bootstrap-servers` son una lista inicial de uno o más intermediarios para que el productor pueda descubrir el resto del clúster. Las propiedades del `serializer` dicen a Kafka cómo deben codificarse la clave y el valor del mensaje. Aquí, le enviaremos mensajes de cadena. Aunque no es obligatorio, siempre se recomienda establecer un `client.id` : esto le permite correlacionar fácilmente las solicitudes en el agente con la instancia del cliente que lo creó.

Otras propiedades interesantes son:

```
props.put("acks", "all");
props.put("retries", 0);
props.put("batch.size", 16384);
props.put("linger.ms", 1);
props.put("buffer.memory", 33554432);
```

Puede controlar la *durabilidad de los mensajes* escritos en Kafka a través de la configuración de `acks` . El valor predeterminado de "1" requiere un reconocimiento explícito del líder de la partición de que la escritura se realizó correctamente. La garantía más `acks=all` que ofrece Kafka es con `acks=all` , lo que garantiza que el líder de la partición no solo aceptó la escritura, sino que se replicó con éxito en todas las réplicas sincronizadas. También puede usar un valor de "0" para maximizar el rendimiento, pero no tendrá ninguna garantía de que el mensaje se haya escrito correctamente en el registro del agente, ya que el agente ni siquiera envía una respuesta en este caso.

`retries` (predeterminados a> 0) determinan si el productor intenta reenviar el mensaje después de una falla. Tenga en cuenta que con los reintentos> 0, la reordenación de mensajes puede ocurrir ya que el reintento puede ocurrir después de que una escritura siguiente haya tenido éxito.

Los productores de Kafka intentan recopilar los mensajes enviados en lotes para mejorar el rendimiento. Con el cliente Java, puede usar `batch.size` para controlar el tamaño máximo en bytes de cada lote de mensajes. Para dar más tiempo para que se llenen los lotes, puede usar `linger.ms` para que el productor retrase el envío. Finalmente, la compresión se puede habilitar con la configuración de tipo de `compression.type` .

Use `buffer.memory` para limitar la memoria total que está disponible para el cliente Java para recopilar mensajes no enviados. Cuando se `max.block.ms` este límite, el productor bloqueará los envíos adicionales durante el tiempo `max.block.ms` antes de generar una excepción. Además, para evitar que los registros se pongan en cola indefinidamente, puede establecer un tiempo de espera con `request.timeout.ms` .

La lista completa de propiedades está disponible aquí . Sugiero leer este artículo de Confluent para más detalles.

Enviando mensajes

El método `send()` es asíncrono. Cuando se llama, agrega el registro a un búfer de envíos de registros pendientes y se devuelve inmediatamente. Esto permite al productor agrupar registros individuales para la eficiencia.

El resultado del envío es un `RecordMetadata` especifica la partición a la que se envió el registro y el desplazamiento que se le asignó. Dado que la llamada de envío es asíncrona, devuelve un `Future` para los Datos de Registro que se asignarán a este registro. Para consultar los metadatos, puede llamar a `get()` , que se bloqueará hasta completar la solicitud o usar una devolución de llamada.

```
// synchronous call with get()
RecordMetadata recordMetadata = producer.send( message ).get();
// callback with a lambda
producer.send( message, ( recordMetadata, error ) -> System.out.println(recordMetadata) );
```

El código

```
public class SimpleProducer{

    public static void main( String[] args ) throws ExecutionException, InterruptedException{
        Properties props = new Properties();

        props.put("bootstrap.servers", "localhost:9092");
        props.put("acks", "all");
        props.put("retries", 0);
        props.put("batch.size", 16384);
        props.put("linger.ms", 1);
        props.put("buffer.memory", 33554432);
        props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
        props.put("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
        props.put( "client.id", "octopus" );

        String topic = "test-topic";

        Producer<String, String> producer = new KafkaProducer<>( props );

        for( int i = 0; i < 10; i++ ){
            ProducerRecord<String, String> message = new ProducerRecord<>( topic, "this is
message " + i );
            producer.send( message );
            System.out.println("message sent.");
        }

        producer.close(); // don't forget this
    }
}
```

Lea Productor / Consumidor en Java en línea: https://riptutorial.com/es/apache-kafka/topic/8974/productor---consumidor-en-java

Capítulo 5: Serializador / Deserializador personalizado

Introducción

Kafka almacena y transporta matrices de bytes en su cola. Los (des) serializadores son responsables de traducir entre la matriz de bytes proporcionada por Kafka y POJOs.

Sintaxis

- public void configure (Map <String,?> config, boolean isKey);
- deserializar T público (String topic, byte [] bytes);
- byte público [] serialize (String topic, T obj);

Parámetros

parámetros	detalles
configuración	las propiedades de configuración (`Properties`) pasadas al `Producer` o al `Consumer` momento de la creación, como un mapa. Contiene configuraciones kafka regulares, pero también puede aumentarse con la configuración definida por el usuario. Es la mejor manera de pasar argumentos al (des) serializador.
es clave	Los (des) serializadores personalizados pueden utilizarse para claves y / o valores. Este parámetro le dice con cuál de los dos tratará esta instancia.
tema	El tema del mensaje actual. Esto le permite definir una lógica personalizada basada en el tema de origen / destino.
bytes	El mensaje en bruto para deserializar.
obj	El mensaje a serializar. Su clase real depende de su serializador.

Observaciones

Antes de la versión 0.9.0.0, la API de Kafka Java utilizaba `Encoders` y `Decoders` . Han sido reemplazados por `Serializer` y `Deserializer` en la nueva API.

Examples

Gson (de) serializador

Este ejemplo utiliza la biblioteca gson para asignar objetos java a cadenas json. Los (des) serializadores son genéricos, ¡pero no siempre tienen que serlo!

Serializador

Código

```
public class GsonSerializer<T> implements Serializer<T> {

    private Gson gson = new GsonBuilder().create();

    @Override
    public void configure(Map<String, ?> config, boolean isKey) {
        // this is called right after construction
        // use it for initialisation
    }

    @Override
    public byte[] serialize(String s, T t) {
        return gson.toJson(t).getBytes();
    }

    @Override
    public void close() {
        // this is called right before destruction
    }
}
```

Uso

Los serializadores se definen a través de las `key.serializer` requeridas `key.serializer` y `value.serializer` productor.

Supongamos que tenemos una clase POJO llamada `SensorValue` y que queremos generar mensajes sin ninguna clave (claves configuradas en `null`):

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
// ... other producer properties ...
props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
props.put("value.serializer", GsonSerializer.class.getName());

Producer<String, SensorValue> producer = new KafkaProducer<>(properties);
// ... produce messages ...
producer.close();
```

(`key.serializer` es una configuración requerida. Como no especificamos las claves de los mensajes, mantenemos el `StringSerializer` entrega con kafka, que puede manejar el `null`).

deserializador

Código

```
public class GsonDeserializer<T> implements Deserializer<T> {

    public static final String CONFIG_VALUE_CLASS = "value.deserializer.class";
    public static final String CONFIG_KEY_CLASS = "key.deserializer.class";
    private Class<T> cls;

    private Gson gson = new GsonBuilder().create();


    @Override
    public void configure(Map<String, ?> config, boolean isKey) {
        String configKey = isKey ? CONFIG_KEY_CLASS : CONFIG_VALUE_CLASS;
        String clsName = String.valueOf(config.get(configKey));

        try {
            cls = (Class<T>) Class.forName(clsName);
        } catch (ClassNotFoundException e) {
            System.err.printf("Failed to configure GsonDeserializer. " +
                    "Did you forget to specify the '%s' property ?%n",
                    configKey);
        }
    }


    @Override
    public T deserialize(String topic, byte[] bytes) {
        return (T) gson.fromJson(new String(bytes), cls);
    }


    @Override
    public void close() {}
}
```

Uso

Los deserializadores se definen a través de las `key.deserializer` requeridas de consumidor `key.deserializer` y `value.deserializer` .

Supongamos que tenemos una clase POJO llamada `SensorValue` y que queremos generar mensajes sin ninguna clave (claves configuradas en `null`):

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
// ... other consumer properties ...
props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer", GsonDeserializer.class.getName());
props.put(GsonDeserializer.CONFIG_VALUE_CLASS, SensorValue.class.getName());

try (KafkaConsumer<String, SensorValue> consumer = new KafkaConsumer<>(props)) {
```

```
    // ... consume messages ...
}
```

Aquí, agregamos una propiedad personalizada a la configuración del consumidor, a saber, `CONFIG_VALUE_CLASS` . El `GsonDeserializer` va a usar en el `configure()` método para determinar qué clase POJO que debe manejar (todas las propiedades añadidas a `props` se pasará a la `configure` método en la forma de un mapa).

Lea Serializador / Deserializador personalizado en línea: https://riptutorial.com/es/apache-kafka/topic/8992/serializador---deserializador-personalizado

Creditos

S. No	Capítulos	Contributors
1	Empezando con apache-kafka	Ali786, Community, Derlin, Laurel, Mandeep Lohan, Matthias J. Sax, Mincong Huang, NangSaigon, Vivek
2	Grupos de consumidores y gestión de compensaciones	Matthias J. Sax, Sönke Liebau
3	herramientas de consola kafka	Derlin
4	Productor / Consumidor en Java	Derlin, ha9u63ar
5	Serializador / Deserializador personalizado	Derlin, G McNicol