EBook Gratuito

APPRENDIMENTO apache-kafka

Free unaffiliated eBook created from
Stack Overflow contributors.

#apache-kafka

Sommario

Di

Capitolo 1: Iniziare con apache-kafka

Osservazioni

Kafka è un sistema di messaggistica di pubblicazione-sottoscrizione high throughput implementato come servizio di registro di commit distribuito, partizionato e replicato.

Tratto dal sito ufficiale di Kafka

> **Veloce**
>
> Un singolo broker Kafka può gestire centinaia di megabyte di letture e scritture al secondo da migliaia di client.
>
> **Scalabile**
>
> Kafka è progettato per consentire a un singolo cluster di fungere da backbone centrale dei dati per una grande organizzazione. Può essere espanso elasticamente e in modo trasparente senza tempi di fermo. I flussi di dati sono partizionati e distribuiti su un cluster di macchine per consentire flussi di dati più grandi delle capacità di una singola macchina e per consentire cluster di consumatori coordinati
>
> **Durevole**
>
> I messaggi sono persistenti su disco e replicati all'interno del cluster per prevenire la perdita di dati. Ogni broker può gestire terabyte di messaggi senza impatto sulle prestazioni.
>
> **Distribuito da Design**
>
> Kafka ha un design moderno incentrato sul cluster che offre una lunga durata e garanzie di tolleranza agli errori.

Examples

Installazione o configurazione

Passaggio 1 . Installa Java 7 o 8

Passaggio 2 . Scarica Apache Kafka all'indirizzo: http://kafka.apache.org/downloads.html

Ad esempio, proveremo a scaricare Apache Kafka 0.10.0.0

Passaggio 3 . Estrai il file compresso.

Su Linux:

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

Sulla finestra: clic destro -> Estrai qui

Passaggio 4 . Avvia Zookeeper

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

Linux:

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

Finestre:

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

Passaggio 5 . Avvia il server Kafka

Linux:

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

Finestre:

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

introduzione

Apache Kafka ™ è una piattaforma di streaming distribuita.

Che significa

1-Ti consente di pubblicare e iscriverti a stream di record. Sotto questo aspetto è simile a una coda di messaggi o sistema di messaggistica aziendale.

2-Consente di archiviare flussi di record in modalità fault-tolerant.

3-Consente di elaborare flussi di record nel momento in cui si verificano.

Viene utilizzato per due ampie classi di applicazioni:

1-Costruire pipeline di dati di streaming in tempo reale che ottengono dati in modo affidabile tra sistemi o applicazioni

2-Creazione di applicazioni di streaming in tempo reale che trasformano o reagiscono ai flussi di dati

> Gli script di console di Kafka sono diversi per piattaforme basate su Unix e Windows. Negli esempi, potrebbe essere necessario aggiungere l'estensione in base alla piattaforma. Linux: script situati in `bin/` con estensione `.sh` . Windows: script situati in `bin\windows\` e con estensione `.bat` .

Installazione

Passaggio 1: scaricare il codice e decomprimerlo:

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

Passaggio 2: avviare il server.

> per poter eliminare gli argomenti in un secondo momento, apri `server.properties` e imposta `delete.topic.enable` su true.

Kafka fa molto affidamento su Zookeeper, quindi è necessario avviarlo prima. Se non lo si è installato, è possibile utilizzare lo script di convenienza fornito con kafka per ottenere un'istanza ZooKeeper a nodo singolo rapida e sporca.

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

Passaggio 3: assicurarsi che tutto sia in esecuzione

Ora dovresti avere zookeeper che ascolta `localhost:2181` e un singolo broker kafka su `localhost:6667` .

Crea un argomento

Abbiamo solo un broker, quindi creiamo un argomento senza fattore di replica e solo una partizione:

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

Controlla il tuo argomento:

```
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
```

inviare e ricevere messaggi

Lancia un consumatore:

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

Su un altro terminale, avviare un produttore e inviare alcuni messaggi. Per impostazione predefinita, lo strumento invia ogni riga come un messaggio separato al broker, senza codifica speciale. Scrivi alcune linee ed esci con CTRL + D o CTRL + C:

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

I messaggi dovrebbero apparire nel terminale del consumatore.

Smetti di kafka

```
kafka-server-stop
```

avviare un cluster multi-broker

Gli esempi precedenti utilizzano solo un broker. Per configurare un cluster reale, abbiamo solo bisogno di avviare più di un server kafka. Si coordineranno automaticamente da soli.

Passo 1: per evitare collisioni, creiamo un file `server.properties` per ogni broker e modifichiamo le proprietà di configurazione `id` , `port` e `logfile` .

Copia:

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

Modifica le proprietà per ogni file, ad esempio:

```
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
```

Step 2: avvia i tre broker:

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

Creare un argomento replicato

```
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
```

Questa volta, ci sono più informazioni:

- "leader" è il nodo responsabile di tutte le letture e scritture per la partizione specificata. Ogni nodo sarà il leader per una porzione selezionata a caso delle partizioni.
- "repliche" è l'elenco di nodi che replicano il log per questa partizione indipendentemente dal fatto che siano il leader o anche se siano attualmente vivi.
- "isr" è l'insieme delle repliche "in-sync". Questo è il sottoinsieme dell'elenco delle repliche che è attualmente vivo e catturato dal leader.

Si noti che l'argomento creato in precedenza è rimasto invariato.

test di tolleranza d'errore

Pubblica un messaggio sul nuovo argomento:

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

Uccidi il leader (1 nel nostro esempio). Su Linux:

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

Su Windows:

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

Guarda cosa è successo:

```
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
```

La leadership è passata al broker 2 e "1" non più in-sync. Ma i messaggi sono ancora lì (usa il consumatore per controllare da solo).

Pulire

Elimina i due argomenti usando:

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

Leggi Iniziare con apache-kafka online: https://riptutorial.com/it/apache-kafka/topic/1986/iniziare-con-apache-kafka

Capitolo 2: Gruppi di consumatori e gestione degli offset

Parametri

Parametro	Descrizione
group.id	Il nome del gruppo di consumatori.
enable.auto.commit	Commettere automaticamente offset; *impostazione predefinita: true* .
auto.commit.interval.ms	Il ritardo minimo in millisecondi tra le commit (richiede `enable.auto.commit=true`); *impostazione predefinita: 5000* .
auto.offset.reset	Cosa fare quando non è stato trovato alcun offset valido commesso; *default: ultimo* . (+)
(+) Valori possibili	**Descrizione**
più presto	Ripristina automaticamente l'offset sul primo offset.
più recente	Ripristina automaticamente l'offset sull'offset più recente.
nessuna	Lanciare un'eccezione al consumatore se non viene trovato nessun offset precedente per il gruppo del consumatore.
qualunque altra cosa	Lanciare un'eccezione al consumatore.

Examples

Cos'è un gruppo di consumatori

A partire da Kafka 0.9, è disponibile il nuovo client di alto livello di KafkaConsumer . Sfrutta un nuovo protocollo Kafka integrato che consente di unire più utenti in un cosiddetto *Consumer Group* . Un gruppo di consumatori può essere descritto come un singolo consumatore logico che si iscrive a una serie di argomenti. Le partions su tutti gli argomenti sono assegnate ai consumatori fisici all'interno del gruppo, in modo tale che ogni patition sia assegnata a un utente esatto (un singolo consumatore può ottenere più partitoni assegnati). I singoli consumatori appartenenti allo stesso gruppo possono essere eseguiti su host diversi in modo distribuito.

I gruppi di consumatori sono identificati tramite il loro `group.id` . Per creare un membro di istanza client specifico di un gruppo di consumatori, è sufficiente assegnare i gruppi `group.id` a questo client, tramite la configurazione del client:

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

Pertanto, tutti i consumatori che si connettono allo stesso cluster Kafka e utilizzano lo stesso `group.id` formano un gruppo di consumatori. I consumatori possono lasciare un gruppo in qualsiasi momento e i nuovi consumatori possono unirsi a un gruppo in qualsiasi momento. In entrambi i casi, viene attivato un cosiddetto *ribilanciamento* e le partizioni vengono riassegnate al Consumer Group per garantire che ciascuna partizione venga elaborata da un solo consumatore all'interno del gruppo.

Presta attenzione, che anche un singolo `KafkaConsumer` forma un gruppo di consumatori con se stesso come membro singolo.

Gestione degli offset dei clienti e tolleranza agli errori

I clienti di Kafka richiedono i messaggi da un broker Kafka tramite una chiamata al `poll()` e il loro avanzamento viene tracciato tramite gli *offset* . Ogni messaggio all'interno di ogni partizione di ogni argomento, ha un cosiddetto offset assegnato, il suo numero di sequenza logica all'interno della partizione. Un `KafkaConsumer` tiene traccia del suo attuale offset per ogni partizione che gli viene assegnata. Presta attenzione, che i broker di Kafka non sono a conoscenza delle attuali compensazioni dei consumatori. Pertanto, nel `poll()` il consumatore deve inviare le sue correzioni correnti al broker, in modo tale che il broker possa restituire i messaggi corrispondenti, cioè ,. messaggi con un offset consecutivo maggiore. Ad esempio, supponiamo di avere un argomento con partizione singola e un singolo utente con offset corrente 5. Nel `poll()` il consumatore invia se offset al broker e il broker restituisce i messaggi per gli offset 6,7,8, ...

Poiché i consumatori tracciano da soli i propri offset, queste informazioni potrebbero andare perse se un consumatore fallisce. Pertanto, gli offset devono essere memorizzati in modo affidabile, in modo tale che al riavvio, un utente possa prelevare il suo vecchio offset e resumer da dove è rimasto. In Kafka, c'è un supporto integrato per questo tramite i *commit di offset* . Il nuovo `KafkaConsumer` può `KafkaConsumer` suo attuale offset a Kafka e Kafka memorizza tali offset in un argomento speciale chiamato `__consumer_offsets` . La memorizzazione degli offset all'interno di un argomento di Kafka non è solo tollerante ai guasti, ma consente anche di riassegnare le partizioni ad altri utenti durante un ribilanciamento. Poiché tutti i consumatori di un gruppo di consumatori possono accedere a tutte le correzioni impegnate di tutte le partizioni, in caso di ribilanciamento, un utente a cui viene assegnata una nuova partizione legge l'offset impegnato di questa partizione `__consumer_offsets` e riprende da dove era rimasto il vecchio utente.

Come commettere offset

KafkaConsumers può eseguire automaticamente il commit degli offset in background (parametro di configurazione `enable.auto.commit = true`) qual è l'impostazione predefinita. Questi commit automatici vengono eseguiti all'interno di `poll()` (che in genere viene chiamato in un ciclo). La frequenza con cui devono essere commessi gli offset, può essere configurata tramite `auto.commit.interval.ms` . Poiché i commit automatici sono incorporati in `poll()` e `poll()` viene chiamato dal codice utente, questo parametro definisce un limite inferiore dell'inter-commit-

interval.

In alternativa al commit automatico, gli offset possono anche essere gestiti manualmente. Per questo, il commit automatico dovrebbe essere disabilitato (`enable.auto.commit = false`). Per il commit manuale `KafkaConsumers` offre due metodi, ovvero commitSync () e commitAsync () . Come indica il nome, `commitSync()` è una chiamata bloccante, che ritorna dopo che gli offset sono stati commessi correttamente, mentre `commitAsync()` restituisce immediatamente. Se vuoi sapere se un commit ha avuto successo o no, puoi fornire un gestore di chiamata (`OffsetCommitCallback`) un parametro di metodo. Prestare attenzione, che in entrambe le chiamate di commit, il consumatore commette gli scostamenti dall'ultima chiamata `poll()` . Per esempio. supponiamo che un argomento di singola partizione con un singolo consumatore e l'ultima chiamata al `poll()` restituisca i messaggi con offset 4,5,6. In caso di commit, l'offset 6 verrà eseguito perché questo è l'ultimo offset tracciato dal cliente consumatore. Allo stesso tempo, sia `commitSync()` che `commitAsync()` consentono un maggiore controllo su quale offset si desidera eseguire il commit: se si utilizzano gli overload corrispondenti che consentono di specificare una `Map<TopicPartition, OffsetAndMetadata>` il consumer impegna solo gli offset specificati (cioè, la mappa può contenere qualsiasi sottoinsieme di partizioni assegnate e l'offset specificato può avere qualsiasi valore).

Semantica di compensazioni commesse

Un offset impegnato indica che tutti i messaggi fino a questo offset sono già stati elaborati. Pertanto, poiché gli offset sono numeri consecutivi, l'offset `x` commette implicitamente tutti gli offset inferiori a `x` Pertanto, non è necessario eseguire il commit di ciascun offset singolarmente e commettere più offset contemporaneamente, ma commettere solo l'offset maggiore.

Prestare attenzione, che in base alla progettazione è anche possibile impegnare un offset minore rispetto all'ultimo offset impegnato. Questo può essere fatto, se i messaggi dovrebbero essere letti una seconda volta.

Garanzie di elaborazione

L'uso del commit automatico fornisce una semantica di elaborazione almeno una volta. L'ipotesi sottostante è che `poll()` viene chiamato solo dopo che tutti i messaggi consegnati in precedenza sono stati elaborati correttamente. Ciò garantisce che nessun messaggio venga perso perché un commit avviene *dopo l'* elaborazione. Se un utente fallisce prima di un commit, tutti i messaggi successivi all'ultimo commit vengono ricevuti da Kafka e processati nuovamente. Tuttavia, questo nuovo tentativo potrebbe comportare duplicati, poiché potrebbe essere stato elaborato un messaggio dall'ultima chiamata `poll()` , ma l'errore si è verificato subito prima della chiamata di commit automatico.

Se sono richieste semantiche di elaborazione al massimo una volta, il commit automatico deve essere disabilitato e deve essere `commitSync()` manualmente un `commitSync()` direttamente dopo il `poll()` . In seguito, i messaggi vengono elaborati. Ciò garantisce che i messaggi vengano inoltrati *prima* che vengano elaborati e quindi non vengano mai letti una seconda volta. Ovviamente, alcuni messaggi potrebbero andare persi in caso di errore.

Come posso leggere l'argomento dall'inizio

Ci sono più strategie per leggere un argomento dall'inizio. Per spiegarli, dobbiamo prima capire cosa succede all'avvio dei consumatori. All'avvio di un consumatore, accade quanto segue:

1. aderire al gruppo di consumatori configurato, che attiva un ribilanciamento e assegna le partizioni al consumatore
2. cerca offset compensati (per tutte le partizioni che sono state assegnate al consumatore)
3. per tutte le partizioni con offset valido, riprendere da questo offset
4. per tutte le partizioni con offset non valido, impostare l'offset iniziale in base al parametro di configurazione `auto.offset.reset`

Inizia un nuovo gruppo di consumatori

Se si desidera elaborare un argomento dall'inizio, è possibile avviare semplicemente un nuovo gruppo di consumatori (ad esempio, selezionare un `group.id` inutilizzato) e impostare `auto.offset.reset = earliest` . Poiché non ci sono offset assegnati per un nuovo gruppo, verrà attivato il ripristino dell'offset automatico e l'argomento verrà utilizzato dall'inizio. Fai attenzione, che al riavvio del consumatore, se usi di nuovo lo stesso `group.id` , non leggerà l'argomento dall'inizio, ma riprenderà da dove è rimasto. Pertanto, per questa strategia, dovrai assegnare un nuovo `group.id` ogni volta che desideri leggere un argomento dall'inizio.

Riutilizzare lo stesso ID di gruppo

Per evitare di impostare un nuovo `group.id` ogni volta che si desidera leggere un argomento dall'inizio, è possibile disabilitare il commit automatico (tramite `enable.auto.commit = false`) prima di avviare il consumer per la prima volta (utilizzando un `group.id` non utilizzato `group.id` e impostazione `auto.offset.reset = earliest`). Inoltre, non si dovrebbe commettere manualmente alcuna compensazione. Poiché gli offset non vengono mai commessi utilizzando questa strategia, al riavvio il consumatore leggerà nuovamente l'argomento dall'inizio.

Tuttavia, questa strategia ha due svantaggi:

1. non è tollerante ai guasti
2. il riequilibrio di gruppo non funziona come previsto

(1) Poiché gli offset non vengono mai commessi, un utente in errore e uno arrestato vengono gestiti allo stesso modo al riavvio. In entrambi i casi, l'argomento verrà consumato dal suo inizio.
(2) Poiché l'offset non viene mai eseguito, sul ribilanciamento le partizioni appena assegnate saranno consumer fin dall'inizio.

Pertanto, questa strategia funziona solo per gruppi di consumatori con un singolo consumatore e deve essere utilizzata solo a fini di sviluppo.

Riutilizza lo stesso ID di gruppo e conferma

Se si desidera essere tolleranti ai guasti e / o utilizzare più utenti nel proprio gruppo di consumatori, è obbligatorio eseguire l'offset. Pertanto, se si desidera leggere un argomento sin dall'inizio, è necessario modificare gli offset commessi all'avvio del consumer. Per questo, `KafkaConsumer` fornisce tre metodi `seek()` , `seekToBeginning()` e `seekToEnd()` . Mentre `seek()` può essere usato per impostare un offset arbitrario, il secondo e il terzo metodo possono essere utilizzati rispettivamente per cercare l'inizio o la fine di una partizione. Pertanto, in caso di fallimento e di ricerca al riavvio del consumatore verrebbe omessa e il consumatore può riprendere da dove è rimasto. Per il consumer-stop-and-restart-from-beginning, `seekToBeginning()` verrebbe chiamato esplicitamente prima di inserire il ciclo di `poll()` . Nota che `seekXXX()` può essere utilizzato solo dopo che un utente è entrato in un gruppo, quindi è necessario eseguire un "dummy-poll" prima di utilizzare `seekXXX()` . Il codice generale sarebbe qualcosa del genere:

```
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
    }
}
```

Leggi Gruppi di consumatori e gestione degli offset online: https://riptutorial.com/it/apache-kafka/topic/5449/gruppi-di-consumatori-e-gestione-degli-offset

Capitolo 3: Produttore / consumatore in Java

introduzione

Questo argomento mostra come produrre e consumare record in Java.

Examples

SimpleConsumer (Kafka> = 0.9.0)

La versione 0.9 di Kafka ha introdotto una riprogettazione completa del consumatore kafka. Se sei interessato al vecchio `SimpleConsumer` (0.8.X), dai un'occhiata a questa pagina . Se l'installazione di Kafka è più recente di 0.8.X, i seguenti codici dovrebbero funzionare immediatamente.

Configurazione e inizializzazione

> Kafka 0.9 non supporta più Java 6 o Scala 2.9. Se si è ancora su Java 6, considerare l'aggiornamento a una versione supportata.

Per prima cosa, crea un progetto maven e aggiungi la seguente dipendenza nel tuo pom:

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

Nota : non dimenticare di aggiornare il campo versione per le ultime versioni (ora> 0.10).

Il consumatore viene inizializzato utilizzando un oggetto `Properties` . Ci sono molte proprietà che ti permettono di mettere a punto il comportamento del consumatore. Di seguito è la configurazione minima necessaria:

```
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());
```

I `bootstrap-servers` sono un elenco iniziale di broker per consentire al consumatore di scoprire il resto del cluster. Non è necessario che siano tutti i server nel cluster: il client determinerà il set completo di broker attivi dai broker in questo elenco.

Il `deserializer` indica al consumatore come interpretare / deserializzare le chiavi e i valori dei

messaggi. Qui, usiamo `StringDeserializer` .

Infine, `group.id` corrisponde al gruppo di consumatori di questo cliente. Ricorda: tutti i consumatori di un gruppo di consumatori divideranno messaggi tra loro (kafka si comporta come una coda di messaggi), mentre i consumatori di diversi gruppi di consumatori riceveranno gli stessi messaggi (kafka si comporta come un sistema di sottoscrizione di pubblicazione).

Altre proprietà utili sono:

- `auto.offset.reset` : controlla cosa fare se l'offset memorizzato in Zookeeper è mancante o fuori range. I valori possibili sono gli `latest` e i `earliest` . Qualsiasi altra cosa genererà un'eccezione;

- `enable.auto.commit` : se `true` (valore predefinito), l'offset del consumatore viene periodicamente (vedere `auto.commit.interval.ms`) salvato in background. Impostandolo su `false` e utilizzando `auto.offset.reset=earliest` - è per determinare da dove deve partire l'utente nel caso in cui non venga trovata alcuna informazione di offset impegnata. mezzi meno `earliest` dall'inizio della partizione argomento assegnata. `latest` mezzo dal più alto numero di offset committed disponibili per la partizione. Tuttavia, il consumer Kafka riprenderà sempre dall'ultimo offset impegnato finché viene trovato un record di offset valido (ovvero ignorando `auto.offset.reset` .) L'esempio migliore è quando un gruppo di consumatori nuovo di zecca si iscrive a un argomento. `auto.offset.reset` per determinare se iniziare dall'inizio (prima) o alla fine (più recente) dell'argomento.

- `session.timeout.ms` : un timeout della sessione garantisce che il blocco venga rilasciato se il consumatore si blocca o se una partizione di rete isola il consumatore dal coordinatore. Infatti:

  > Quando si fa parte di un gruppo di consumatori, a ciascun consumatore viene assegnato un sottoinsieme delle partizioni dagli argomenti a cui è stato abbonato. Questo è fondamentalmente un blocco di gruppo su quelle partizioni. Finché il blocco è trattenuto, nessun altro membro del gruppo sarà in grado di leggere da loro. Quando il tuo consumatore è sano, questo è esattamente quello che vuoi. È l'unico modo per evitare il consumo duplicato. Ma se il consumatore muore a causa di un errore della macchina o dell'applicazione, è necessario che il blocco venga rilasciato in modo che le partizioni possano essere assegnate a un membro sano. fonte

L'elenco completo delle proprietà è disponibile qui http://kafka.apache.org/090/documentation.html#newconsumerconfigs .

Creazione di consumatori e abbonamento per argomento

Una volta che abbiamo le proprietà, creare un consumatore è facile:

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

Dopo esserti iscritto, il consumatore può coordinarsi con il resto del gruppo per ottenere l'assegnazione della partizione. Tutto ciò viene gestito automaticamente quando inizi a consumare dati.

Sondaggio di base

Il consumatore ha bisogno di essere in grado di recuperare i dati in parallelo, potenzialmente da molte partizioni per molti argomenti probabilmente distribuiti su molti broker. Fortunatamente, tutto questo viene gestito automaticamente quando inizi a consumare dati. Per farlo, tutto ciò che devi fare è chiamare il `poll` in un ciclo e il consumatore gestirà il resto.

`poll` restituisce un (forse vuoto) insieme di messaggi dalle partizioni che sono state assegnate.

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

Il codice

Esempio di base

Questo è il codice più semplice che puoi usare per recuperare i messaggi da un argomento di 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 );
            }
        }
    }
}
```

Esempio eseguibile

> Il consumatore è progettato per essere eseguito nella propria thread. Non è sicuro per l'uso con multithreading senza sincronizzazione esterna e probabilmente non è una buona idea da provare.

Di seguito è riportata una semplice attività Runnable che inizializza l'utente, sottoscrive un elenco di argomenti ed esegue il ciclo di polling indefinitamente fino allo spegnimento esterno.

```
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();
    }
}
```

Tieni presente che durante il sondaggio utilizziamo un timeout di `Long.MAX_VALUE` , pertanto attenderà indefinitamente un nuovo messaggio. Per chiudere correttamente il consumatore, è importante chiamare il suo metodo `shutdown()` prima di terminare l'applicazione.

Un driver potrebbe usarlo in questo modo:

```
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)

Configurazione e inizializzazione

Per prima cosa, crea un progetto maven e aggiungi la seguente dipendenza nel tuo pom:

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

Il produttore viene inizializzato utilizzando un oggetto `Properties` . Ci sono molte proprietà che ti permettono di mettere a punto il comportamento del produttore. Di seguito è la configurazione minima necessaria:

```
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");
```

I `bootstrap-servers` sono un elenco iniziale di uno o più broker per consentire al produttore di scoprire il resto del cluster. Le proprietà `serializer` dicono a Kafka come la chiave e il valore del messaggio dovrebbero essere codificati. Qui, invieremo messaggi di stringa. Sebbene non sia richiesto, l'impostazione di un `client.id` è sempre consigliata: ciò consente di correlare facilmente le richieste sul broker con l'istanza client che lo ha creato.

Altre proprietà interessanti sono:

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

Puoi controllare la *durata dei messaggi* scritti su Kafka attraverso l'impostazione degli `acks` . Il valore predefinito di "1" richiede un riconoscimento esplicito dal leader della partizione che la scrittura ha avuto esito positivo. La più forte garanzia offerta da Kafka è `acks=all` , che garantisce che non solo il leader della partizione accetta la scrittura, ma è stata replicata con successo su tutte le repliche in-sync. Puoi anche utilizzare il valore "0" per massimizzare il throughput, ma non avrai alcuna garanzia che il messaggio sia stato scritto correttamente nel log del broker poiché il broker non invia nemmeno una risposta in questo caso.

`retries` (predefinito su> 0) determina se il produttore tenta di inviare nuovamente il messaggio dopo un errore. Si noti che con i tentativi> 0, può verificarsi un riordino dei messaggi poiché il tentativo può verificarsi dopo una successiva scrittura riuscita.

I produttori di Kafka tentano di raccogliere i messaggi inviati in lotti per migliorare il rendimento. Con il client Java, è possibile utilizzare `batch.size` per controllare la dimensione massima in byte di ciascun batch di messaggi. Per dare più tempo per i lotti da riempire, puoi usare `linger.ms` per far ritardare la spedizione al produttore. Infine, la compressione può essere abilitata con l'impostazione `compression.type` .

Utilizzare `buffer.memory` per limitare la memoria totale disponibile al client Java per la raccolta di messaggi non inviati. Quando viene raggiunto questo limite, il produttore bloccherà le mandate aggiuntive fino a `max.block.ms` prima di generare un'eccezione. Inoltre, per evitare di mantenere i record accodati indefinitamente, è possibile impostare un timeout utilizzando `request.timeout.ms` .

L'elenco completo delle proprietà è disponibile qui . Suggerisco di leggere questo articolo di Confluent per maggiori dettagli.

Invio di messaggi

Il metodo `send()` è asincrono. Quando viene chiamato, aggiunge il record a un buffer di record in sospeso inviati e restituisce immediatamente. Ciò consente al produttore di raggruppare i record individuali per l'efficienza.

Il risultato di send è un `RecordMetadata` specifica la partizione su cui è stato inviato il record e l'offset che è stato assegnato. Poiché la chiamata di invio è asincrona, restituisce un `Future` per i RecordMetadata che verranno assegnati a questo record. Per consultare i metadati, è possibile chiamare `get()` , che bloccherà fino a quando la richiesta non verrà completata o non verrà utilizzata una richiamata.

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

Il codice

```
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
    }
}
```

Leggi Produttore / consumatore in Java online: https://riptutorial.com/it/apache-kafka/topic/8974/produttore---consumatore-in-java

Capitolo 4: Serializzatore / deserializzatore personalizzato

introduzione

Kafka memorizza e trasporta gli array di byte nella sua coda. I (de) serializzatori sono responsabili della traduzione tra l'array di byte fornito da Kafka e POJO.

Sintassi

- public void configure (Map <String,?> config, boolean isKey);
- public T deserialize (argomento String, byte [] byte);
- byte pubblico [] serialize (argomento String, T obj);

Parametri

parametri	dettagli
config	le proprietà di configurazione (`Properties`) passate al `Producer` o al `Consumer` momento della creazione, come una mappa. Contiene configurazioni kafka regolari, ma può anche essere ampliato con la configurazione definita dall'utente. È il modo migliore per passare argomenti al (de) serializzatore.
isKey	i serializzatori personalizzati (de) possono essere utilizzati per chiavi e / o valori. Questo parametro indica quale dei due questa istanza tratterà.
argomento	l'argomento del messaggio corrente. Ciò consente di definire la logica personalizzata in base all'argomento sorgente / destinazione.
byte	Il messaggio grezzo da deserializzare
obj	Il messaggio da serializzare. La sua classe effettiva dipende dal tuo serializzatore.

Osservazioni

Prima della versione 0.9.0.0 l'API Java di Kafka utilizzava `Encoders` and `Decoders` . Sono stati sostituiti da `Serializer` e `Deserializer` nella nuova API.

Examples

Gson (de) serializzatore

Questo esempio usa la libreria gson per mappare oggetti java su stringhe json. I (de) serializzatori sono generici, ma non sempre devono essere!

Serializer

Codice

```
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

I serializzatori vengono definiti tramite le proprietà del produttore `key.serializer` e `value.serializer` richieste.

Supponiamo di avere una classe POJO denominata `SensorValue` e che vogliamo produrre messaggi senza alcuna chiave (le chiavi sono impostate su `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` è una configurazione richiesta. Poiché non specifichiamo le chiavi dei messaggi, manteniamo `StringSerializer` fornito con `StringSerializer` , che è in grado di gestire `null`).

deserializzatore

Codice

```
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

I deserializzatori sono definiti attraverso le proprietà necessarie di `key.deserializer` e `value.deserializer` .

Supponiamo di avere una classe POJO denominata `SensorValue` e che vogliamo produrre messaggi senza alcuna chiave (le chiavi sono impostate su `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 ...
}
```

Qui, aggiungiamo una proprietà personalizzata alla configurazione del consumatore, ovvero `CONFIG_VALUE_CLASS` . `GsonDeserializer` lo utilizzerà nel metodo `configure()` per determinare quale classe POJO deve gestire (tutte le proprietà aggiunte agli `props` saranno passate al metodo `configure` sotto forma di una mappa).

Leggi Serializzatore / deserializzatore personalizzato online: https://riptutorial.com/it/apache-kafka/topic/8992/serializzatore---deserializzatore-personalizzato

Capitolo 5: strumenti di console di kafka

introduzione

Kafka offre strumenti da riga di comando per gestire argomenti, gruppi di consumatori, per consumare e pubblicare messaggi e così via.

Importante : gli script della console di Kafka sono diversi per piattaforme basate su Unix e Windows. Negli esempi, potrebbe essere necessario aggiungere l'estensione in base alla piattaforma.

Linux : script situati in `bin/` con estensione `.sh` .

Windows : script situati in `bin\windows\` e con estensione `.bat` .

Examples

Kafka-argomenti

Questo strumento ti consente di elencare, creare, modificare e descrivere argomenti.

Elenco argomenti:

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

Crea un argomento:

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

crea un argomento con una partizione e nessuna replica.

Descrivi un argomento:

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

Modificare un argomento:

```
# 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
```

(Attenzione: Kafka non supporta la riduzione del numero di partizioni di un argomento) (consulta questo elenco di proprietà di configurazione)

Kafka-console-produttore

Questo strumento ti consente di produrre messaggi dalla riga di comando.

Invia messaggi di stringa semplici a un argomento:

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

(ogni nuova riga è un nuovo messaggio, digita ctrl + D o ctrl + C per interrompere)

Invia messaggi con le chiavi:

```
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
```

Invia messaggi da un file:

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

Kafka-console-consumer

Questo strumento ti consente di consumare messaggi da un argomento.

> per usare la vecchia implementazione consumer, sostituire `--bootstrap-server` con `--zookeeper` .

Mostra semplici messaggi:

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

Consuma vecchi messaggi:

Per vedere i vecchi messaggi, puoi usare l'opzione `--from-beginning` .

Visualizza i messaggi valore-chiave :

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

kafka-semplice-consumer-shell

Questo consumatore è uno strumento di basso livello che ti consente di utilizzare messaggi provenienti da partizioni, offset e repliche specifici.

Parametri utili:

- `parition` : la partizione specifica da cui consumare (predefinito a tutti)
- `offset` : l'offset iniziale. Usa `-2` per consumare messaggi dall'inizio, `-1` per consumare alla fine.
- `max-messages` : numero di messaggi da stampare
- `replica` : la replica, predefinita per il broker-leader (-1)

Esempio:

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

visualizza 3 messaggi dalla partizione 1 a partire dall'offset 4 dall'argomento argomento-argomento.

Kafka-consumo-gruppi

Questo strumento ti consente di elencare, descrivere o eliminare gruppi di consumatori. Dai un'occhiata a questo articolo per ulteriori informazioni sui gruppi di consumatori.

> se usi ancora la vecchia implementazione consumer, sostituisci `--bootstrap-server` con `--zookeeper` .

Elenca gruppi di consumatori:

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

Descrivi un gruppo di consumatori:

```
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
```

Note : nell'output sopra,

- `current-offset` è l'ultimo offset impegnato dell'istanza consumer,
- `log-end-offset` è l'offset più alto della partizione (quindi, sommando questa colonna si ottiene il numero totale di messaggi per l'argomento)
- `lag` è la differenza tra l'attuale compensazione del consumatore e l'offset più alto, quindi

quanto è indietro il consumatore,
- `owner` è il `client.id` del consumatore (se non specificato, viene visualizzato uno di default).

Elimina un gruppo di consumatori:

> la cancellazione è disponibile solo quando i metadati di gruppo sono memorizzati in zookeeper (vecchia consumer api). Con la nuova API consumer, il broker gestisce tutto, inclusa l'eliminazione dei metadati: il gruppo viene eliminato automaticamente quando scade l'ultimo offset impegnato per il gruppo.

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

Leggi strumenti di console di kafka online: https://riptutorial.com/it/apache-kafka/topic/8990/strumenti-di-console-di-kafka

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare con apache-kafka	Ali786, Community, Derlin, Laurel, Mandeep Lohan, Matthias J. Sax, Mincong Huang, NangSaigon, Vivek
2	Gruppi di consumatori e gestione degli offset	Matthias J. Sax, Sönke Liebau
3	Produttore / consumatore in Java	Derlin, ha9u63ar
4	Serializzatore / deserializzatore personalizzato	Derlin, G McNicol
5	strumenti di console di kafka	Derlin