eBook Gratuit

APPRENEZ apache-kafka

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#apache-kafka

Table des matières

À propos

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.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

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

Chapitre 1: Démarrer avec apache-kafka

Remarques

Kafka est un système de messagerie de publication / abonnement à haut débit implémenté en tant que service de journal de validation distribué, partitionné et répliqué.

Tiré du site officiel de Kafka

> **Vite**
>
> Un courtier Kafka unique peut gérer des centaines de mégaoctets de lectures et d'écritures par seconde à partir de milliers de clients.
>
> **Évolutive**
>
> Kafka est conçu pour permettre à un cluster unique de servir de réseau central de données pour une grande organisation. Il peut être étendu de manière élastique et transparente sans temps d'arrêt. Les flux de données sont partitionnés et répartis sur un cluster de machines pour permettre des flux de données supérieurs à la capacité d'une machine unique et pour permettre des grappes de consommateurs coordonnés
>
> **Durable**
>
> Les messages sont conservés sur le disque et répliqués dans le cluster pour éviter toute perte de données. Chaque courtier peut gérer des téraoctets de messages sans impact sur les performances.
>
> **Distribué par Design**
>
> Kafka a une conception moderne centrée sur les grappes qui offre une grande durabilité et des garanties de tolérance aux pannes.

Examples

Installation ou configuration

Étape 1 . Installez Java 7 ou 8

Étape 2 . Téléchargez Apache Kafka sur: http://kafka.apache.org/downloads.html

Par exemple, nous allons essayer de télécharger Apache Kafka 0.10.0.0

Étape 3 . Extrayez le fichier compressé.

Sous Linux:

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

On Window: Clic droit -> Extraire ici

Étape 4 . Commencer Zookeeper

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

Linux:

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

Les fenêtres:

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

Étape 5 . Démarrer le serveur Kafka

Linux:

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

Les fenêtres:

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

introduction

Apache Kafka ™ est une plate-forme de diffusion distribuée.

Ce qui signifie

1-Il vous permet de publier et de vous abonner à des flux d’enregistrements. À cet égard, il est similaire à une file d'attente de messages ou à un système de messagerie d'entreprise.

2-It vous permet de stocker des flux d'enregistrements d'une manière tolérante aux pannes.

3-It vous permet de traiter des flux d'enregistrements à mesure qu'ils se produisent.

Il s'utilise pour deux grandes catégories d'applications:

Pipelines de données en temps réel en flux continu permettant de générer des données entre systèmes ou applications

2 applications de streaming en temps réel qui transforment ou réagissent aux flux de données

> Les scripts de console Kafka sont différents pour les plates-formes Unix et Windows.

> Dans les exemples, vous devrez peut-être ajouter l'extension en fonction de votre plate-forme. Linux: scripts situés dans `bin/` avec l'extension `.sh` . Windows: scripts situés dans `bin\windows\` et avec l'extension `.bat` .

Installation

Étape 1: Téléchargez le code et décompressez-le:

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

Étape 2: démarrez le serveur.

> pour pouvoir supprimer des rubriques ultérieurement, ouvrez `server.properties` et définissez `delete.topic.enable` sur true.

Kafka s'appuie fortement sur zookeeper, vous devez donc commencer par le début. Si vous ne l'avez pas installé, vous pouvez utiliser le script de commodité fourni avec kafka pour obtenir une instance ZooKeeper à noeud unique rapide et sale.

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

Étape 3: assurez-vous que tout fonctionne bien

Zookeeper devrait maintenant écouter `localhost:2181` et un seul courtier kafka sur `localhost:6667` .

Créer un sujet

Nous n'avons qu'un seul courtier, nous créons donc un sujet sans facteur de réplication et une seule partition:

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

Vérifiez votre sujet:

```
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
```

envoyer et recevoir des messages

Lancer un consommateur:

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

Sur un autre terminal, lancez un producteur et envoyez des messages. Par défaut, l'outil envoie chaque ligne en tant que message distinct au courtier, sans codage spécial. Ecrivez des lignes et quittez avec CTRL + D ou CTRL + C:

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

Les messages doivent apparaître dans le therminal du consommateur.

Arrêter kafka

```
kafka-server-stop
```

démarrer un cluster multi-courtier

Les exemples ci-dessus utilisent un seul courtier. Pour configurer un vrai cluster, il suffit de démarrer plusieurs serveurs kafka. Ils se coordonneront automatiquement.

Etape 1: pour éviter les collisions, nous créons un fichier `server.properties` pour chaque courtier et modifions les propriétés de configuration `id` , `port` et `logfile` .

Copie:

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

Modifier les propriétés de chaque fichier, par exemple:

```
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
```

Etape 2: lancez les trois courtiers:

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

Créer un sujet répliqué

```
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
```

Cette fois, il y a plus d'informations:

- "leader" est le noeud responsable de toutes les lectures et écritures pour la partition donnée. Chaque nœud sera le leader d'une partie des partitions sélectionnée de manière aléatoire.
- "répliques" est la liste des noeuds qui répliquent le journal pour cette partition, qu'ils soient le leader ou même s'ils sont actuellement actifs.
- "isr" est l'ensemble des répliques "in-sync". Ceci est le sous-ensemble de la liste de réplicas qui est actuellement en vie et rattrapé par le leader.

Notez que le sujet précédemment créé reste inchangé.

test de tolérance aux pannes

Publier un message sur le nouveau sujet:

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

Tuez le chef (1 dans notre exemple). Sous Linux:

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

Sous Windows:

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

Voir ce qui s'est passé:

```
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 direction est passée au courtier 2 et le "1" n'est plus synchronisé. Mais les messages sont toujours là (utilisez le consommateur pour vérifier par vous-même).

Nettoyer

Supprimez les deux sujets en utilisant:

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

Lire Démarrer avec apache-kafka en ligne: https://riptutorial.com/fr/apache-kafka/topic/1986/demarrer-avec-apache-kafka

Chapitre 2: Groupes de consommateurs et gestion des compensations

Paramètres

Paramètre	La description
group.id	Le nom du groupe de consommateurs.
enable.auto.commit	Commettre automatiquement des compensations; *default: true* .
auto.commit.interval.ms	Le délai minimum en millisecondes entre les `enable.auto.commit=true` (nécessite `enable.auto.commit=true`); *par défaut: 5000* .
auto.offset.reset	Que faire en l'absence de validation validée validée? *par défaut: le plus récent* (+)
(+) Valeurs possibles	**La description**
au plus tôt	Réinitialise automatiquement le décalage au décalage le plus précoce.
dernier	Rétablir automatiquement le décalage au dernier décalage.
aucun	Jetez une exception au consommateur si aucun décalage précédent n'a été trouvé pour le groupe du consommateur.
rien d'autre	Jetez une exception au consommateur.

Examples

Qu'est-ce qu'un groupe de consommateurs?

À partir de Kafka 0.9, le nouveau client de haut niveau KafkaConsumer est disponible. Il exploite un nouveau protocole Kafka intégré qui permet de combiner plusieurs consommateurs dans un groupe appelé *Consumer Group* . Un groupe de consommateurs peut être décrit comme un consommateur logique unique qui souscrit à un ensemble de rubriques. Les partions de tous les sujets sont attribuées aux consommateurs physiques du groupe, de sorte que chaque demande est attribuée à un seul consommateur (un seul consommateur peut recevoir plusieurs parties attribuées). Les consommateurs individuels appartenant au même groupe peuvent fonctionner sur des hôtes différents de manière distribuée.

Les groupes de consommateurs sont identifiés via leur `group.id` . Pour créer un membre

d'instance client spécifique d'un groupe de consommateurs, il suffit d'affecter les groupes `group.id` à ce client, via la configuration du client:

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

Ainsi, tous les consommateurs qui se connectent au même cluster Kafka et utilisent le même `group.id` forment un groupe de consommateurs. Les consommateurs peuvent quitter un groupe à tout moment et les nouveaux consommateurs peuvent rejoindre un groupe à tout moment. Dans les deux cas, un soi-disant *rééquilibrage* est déclenché et les partitions sont réaffectées au groupe de consommateurs pour garantir que chaque partition est traitée par un seul consommateur du groupe.

Faites attention, même un seul `KafkaConsumer` forme un groupe de consommateurs en tant que membre unique.

Gestion des crédits à la consommation et tolérance aux fautes

KafkaConsumers demande des messages à un courtier Kafka via un appel à `poll()` et leur progression est suivie par des *décalages* . Chaque message de chaque partition de chaque sujet est associé à un décalage, c'est-à-dire son numéro de séquence logique dans la partition. Un `KafkaConsumer` suit son décalage actuel pour chaque partition qui lui est affectée. Faites attention, les courtiers Kafka ne sont pas au courant des compensations actuelles des consommateurs. Ainsi, sur `poll()` le consommateur doit envoyer ses décalages actuels au courtier, de sorte que le courtier puisse renvoyer les messages correspondants, c.-à-d. messages avec un décalage consécutif plus important. Par exemple, supposons que nous ayons un seul sujet de partition et un seul consommateur avec le décalage actuel 5. Sur `poll()` le consommateur envoie le décalage au courtier et le courtier renvoie les messages pour les décalages 6,7,8, ...

Étant donné que les consommateurs suivent eux-mêmes leurs décalages, ces informations peuvent être perdues en cas de défaillance d'un consommateur. Par conséquent, les décalages doivent être stockés de manière fiable, de sorte qu'au redémarrage, un consommateur puisse récupérer son ancien décalage et son nouvel article là où il l'a laissé. Dans Kafka, il existe un support intégré pour cela via des *commits offset* . Le nouveau `KafkaConsumer` peut valider son décalage actuel sur Kafka et Kafka stocke ces décalages dans une rubrique spéciale appelée `__consumer_offsets` . Stocker les décalages dans une rubrique Kafka n'est pas seulement tolérant aux pannes, mais permet également de réaffecter des partitions à d'autres consommateurs lors d'un rééquilibrage. Étant donné que tous les consommateurs d'un groupe de consommateurs peuvent accéder à tous les décalages validés de toutes les partitions, lors d'un rééquilibrage, un consommateur qui reçoit une nouvelle partition lit simplement le décalage `__consumer_offsets` de cette partition à partir de la rubrique `__consumer_offsets` .

Comment commettre des compensations

KafkaConsumers peut valider automatiquement les décalages en arrière-plan (paramètre de configuration `enable.auto.commit = true`) quel est le paramètre par défaut. Ces validations

automatiques sont effectuées dans `poll()` (généralement appelé dans une boucle). La fréquence à laquelle les décalages doivent être `auto.commit.interval.ms` peut être configurée via `auto.commit.interval.ms` . Comme les validations automatiques sont incorporées dans `poll()` et que `poll()` est appelé par le code utilisateur, ce paramètre définit une limite inférieure pour l'intervalle inter-validation.

Comme alternative à la validation automatique, les décalages peuvent également être gérés manuellement. Pour cela, la validation automatique doit être désactivée (`enable.auto.commit = false`). Pour la `KafkaConsumers` manuelle, `KafkaConsumers` propose deux méthodes, à savoir commitSync () et commitAsync () . Comme son nom l'indique, `commitSync()` est un appel bloquant qui retourne après que les décalages ont été `commitAsync()` , alors que `commitAsync()` retourne immédiatement. Si vous voulez savoir si une validation a réussi ou non, vous pouvez fournir un gestionnaire de `OffsetCommitCallback` (`OffsetCommitCallback`) à un paramètre de méthode. Faites attention, dans les deux appels de validation, le consommateur valide les décalages du dernier appel `poll()` . Par exemple. supposons un sujet de partition unique avec un seul consommateur et le dernier appel à `poll()` renvoie des messages avec des décalages 4,5,6. Lors de la validation, le décalage 6 sera validé car il s'agit du dernier décalage suivi par le client consommateur. En même temps, `commitSync()` et `commitAsync()` permettent plus de contrôle sur le décalage que vous souhaitez valider: si vous utilisez les surcharges correspondantes vous permettant de spécifier une `Map<TopicPartition, OffsetAndMetadata>` le consommateur ne `Map<TopicPartition, OffsetAndMetadata>` que les décalages spécifiés (c.-à-d. que la carte peut contenir n'importe quel sous-ensemble de partitions assignées et que le décalage spécifié peut avoir n'importe quelle valeur).

Sémantique des compensations engagées

Un décalage engagé indique que tous les messages jusqu'à ce décalage ont déjà été traités. Ainsi, comme les décalages sont des nombres consécutifs, la validation du décalage `x` valide implicitement tous les décalages inférieurs à `x` Par conséquent, il n'est pas nécessaire de valider chaque décalage individuellement et de commettre plusieurs décalages à la fois, mais en validant le plus grand décalage.

Faites attention, car de par sa conception, il est également possible de commettre un décalage plus petit que le dernier décalage engagé. Cela peut être fait si les messages doivent être lus une seconde fois.

Traitement des garanties

L'utilisation de la validation automatique fournit une sémantique de traitement au moins une fois. L'hypothèse sous-jacente est que `poll()` est uniquement appelé après que tous les messages précédemment livrés ont été traités avec succès. Cela garantit qu'aucun message ne soit perdu car une validation se produit *après le* traitement. Si un consommateur tombe en panne avant une validation, tous les messages après la dernière validation sont reçus de Kafka et traités à nouveau. Cependant, cette tentative peut entraîner des doublons, car certains messages du dernier appel `poll()` ont peut-être été traités, mais l'échec s'est produit juste avant l'appel de validation automatique.

Si une sémantique de traitement au plus une fois est requise, la validation automatique doit être désactivée et un `commitSync()` manuel directement après `poll()` doit être effectué. Par la suite, les messages sont traités. Cela garantit que les messages sont validés *avant* leur traitement et ne sont donc jamais lus une seconde fois. Bien sûr, certains messages peuvent être perdus en cas d'échec.

Comment puis-je lire le sujet depuis ses débuts

Il existe plusieurs stratégies pour lire un sujet depuis ses débuts. Pour les expliquer, nous devons d'abord comprendre ce qui se passe au démarrage du consommateur. Au démarrage d'un consommateur, les événements suivants se produisent:

1. rejoindre le groupe de consommateurs configuré, ce qui déclenche un rééquilibrage et attribue des partitions au consommateur
2. rechercher les compensations validées (pour toutes les partitions affectées au consommateur)
3. pour toutes les partitions avec un offset valide, reprendre à partir de ce décalage
4. pour toutes les partitions avec décalage non valide, définissez le décalage de départ en fonction du paramètre de configuration `auto.offset.reset`

Démarrer un nouveau groupe de consommateurs

Si vous souhaitez traiter un sujet depuis son début, vous pouvez simplement démarrer un nouveau groupe de consommateurs (par exemple, choisir un `group.id` inutilisé) et définir `auto.offset.reset = earliest` . Comme il n'y a pas de décalages validés pour un nouveau groupe, la réinitialisation automatique du décalage sera déclenchée et le sujet sera utilisé dès le début. Faites attention, au redémarrage du consommateur, si vous utilisez à nouveau le même `group.id` , il ne lira pas le sujet de nouveau, mais reprendra où il est parti. Ainsi, pour cette stratégie, vous devrez attribuer un nouveau `group.id` chaque fois que vous souhaitez lire un sujet depuis le début.

Réutiliser le même identifiant de groupe

Pour éviter de définir un nouveau `group.id` chaque fois que vous souhaitez lire un sujet depuis son début, vous pouvez désactiver la validation automatique (via `enable.auto.commit = false`) avant de lancer le consommateur pour la toute première fois (en utilisant un `group.id` inutilisé `group.id` et paramètre `auto.offset.reset = earliest`). De plus, vous ne devez pas commettre de décalage manuellement. Comme les décalages ne sont jamais validés avec cette stratégie, au redémarrage, le consommateur relira le sujet depuis le début.

Cependant, cette stratégie présente deux inconvénients:

1. ce n'est pas tolérant aux fautes
2. le rééquilibrage du groupe ne fonctionne pas comme prévu

(1) Comme les décalages ne sont jamais validés, un consommateur défaillant et arrêté est traité de la même manière au redémarrage. Dans les deux cas, le sujet sera consommé dès le début.
(2) Le décalage n'étant jamais validé, lors du rééquilibrage, les partitions nouvellement assignées

seront consommées dès le début.

Par conséquent, cette stratégie ne fonctionne que pour les groupes de consommateurs avec un seul consommateur et ne devrait être utilisée qu'à des fins de développement.

Réutiliser le même identifiant de groupe et la même validation

Si vous souhaitez être tolérant aux pannes et / ou utiliser plusieurs consommateurs dans votre groupe de consommateurs, la validation des compensations est obligatoire. Ainsi, si vous souhaitez lire un sujet depuis le début, vous devez manipuler les décalages validés au démarrage du consommateur. Pour cela, `KafkaConsumer` fournit trois méthodes `seek()` , `seekToBeginning()` et `seekToEnd()` . Alors que `seek()` peut être utilisé pour définir un décalage arbitraire, les deuxième et troisième méthodes peuvent être utilisées pour rechercher respectivement le début et la fin d'une partition. Ainsi, en cas d'échec et de redémarrage, la recherche de redémarrage serait omise et le consommateur pourrait reprendre sa sortie. Pour les utilisateurs de stop-and-restart-from-`seekToBeginning()` , `seekToBeginning()` serait appelé explicitement avant d'entrer dans votre boucle `poll()` . Notez que `seekXXX()` ne peut être utilisé qu'après qu'un consommateur a rejoint un groupe - il est donc nécessaire d'effectuer un "sondage factice" avant d'utiliser `seekXXX()` . Le code global serait quelque chose comme ceci:

```
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
    }
}
```

Lire Groupes de consommateurs et gestion des compensations en ligne: https://riptutorial.com/fr/apache-kafka/topic/5449/groupes-de-consommateurs-et-gestion-des-compensations

Chapitre 3: les outils de la console kafka

Introduction

Kafka propose des outils en ligne de commande pour gérer des sujets, des groupes de consommateurs, pour consommer et publier des messages, etc.

Important : les scripts de la console Kafka sont différents pour les plates-formes Unix et Windows. Dans les exemples, vous devrez peut-être ajouter l'extension en fonction de votre plate-forme.

Linux : scripts situés dans `bin/` avec l'extension `.sh` .

Windows : scripts situés dans `bin\windows\` et avec l'extension `.bat` .

Examples

kafka-topics

Cet outil vous permet de lister, créer, modifier et décrire des sujets.

Liste des sujets:

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

Créer un sujet:

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

crée un sujet avec une partition et aucune réplication.

Décrivez un sujet:

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

Modifier un sujet:

```
# 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
```

(Attention: Kafka ne prend pas en charge la réduction du nombre de partitions d'un sujet) (voir cette liste de propriétés de configuration)

kafka-console-producteur

Cet outil vous permet de produire des messages à partir de la ligne de commande.

Envoyer des messages de chaîne simples à un sujet:

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

(chaque nouvelle ligne est un nouveau message, tapez ctrl + D ou ctrl + C pour arrêter)

Envoyer des messages avec les clés:

```
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
```

Envoyer des messages depuis un fichier:

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

kafka-console-consumer

Cet outil vous permet de consommer des messages d'un sujet.

> pour utiliser l'ancienne implémentation du consommateur, remplacez `--bootstrap-server` par `--zookeeper` .

Afficher des messages simples:

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

Consommez d'anciens messages:

Pour voir les anciens messages, vous pouvez utiliser l'option `--from-beginning` .

Afficher les messages de valeur-clé :

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

kafka-simple-consommateur-shell

Ce consommateur est un outil de bas niveau qui vous permet de consommer des messages à partir de partitions, de décalages et de répliques spécifiques.

Paramètres utiles:

- `parition` : la partition spécifique à `parition` (par défaut à tous)
- `offset` : le décalage de début. Utilisez `-2` pour consommer des messages depuis le début, `-1` pour consommer depuis la fin.
- `max-messages` : nombre de messages à imprimer
- `replica` : la réplique, par défaut pour le courtier leader (-1)

Exemple:

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

affiche 3 messages de la partition 1 commençant à l'offset 4 du sujet de test du sujet.

kafka-groupes de consommateurs

Cet outil vous permet de répertorier, décrire ou supprimer des groupes de consommateurs. Consultez cet article pour plus d'informations sur les groupes de consommateurs.

> Si vous utilisez toujours l'ancienne implémentation du consommateur, remplacez `--bootstrap-server` par `--zookeeper` .

Liste des groupes de consommateurs:

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

Décrivez un groupe de consommateurs:

```
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
```

Remarques : dans la sortie ci-dessus,

- `current-offset` est le dernier décalage engagé de l'instance consommateur,
- `log-end-offset` est le `log-end-offset` le plus élevé de la partition (par conséquent, la somme de cette colonne vous donne le nombre total de messages pour le sujet)
- `lag` est la différence entre le décalage actuel du consommateur et le décalage le plus élevé, d'où la distance

- `owner` est le `client.id` du consommateur (s'il n'est pas spécifié, un par défaut est affiché).

Supprimer un groupe de consommateurs:

> la suppression n'est disponible que lorsque les métadonnées du groupe sont stockées dans zookeeper (ancienne interface client). Avec la nouvelle API du consommateur, le courtier gère tout, y compris la suppression des métadonnées: le groupe est automatiquement supprimé lorsque le dernier décalage validé du groupe expire.

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

Lire les outils de la console kafka en ligne: https://riptutorial.com/fr/apache-kafka/topic/8990/les-outils-de-la-console-kafka

Chapitre 4: Producteur / Consommateur en Java

Introduction

Cette rubrique montre comment produire et consommer des enregistrements en Java.

Examples

SimpleConsumer (Kafka> = 0.9.0)

La version 0.9 de Kafka a introduit une refonte complète du consommateur de kafka. Si vous êtes intéressé par l'ancien `SimpleConsumer` (0.8.X), consultez cette page . Si votre installation de Kafka est plus récente que la 0.8.X, les codes suivants devraient être intégrés.

Configuration et initialisation

> Kafka 0.9 ne prend plus en charge Java 6 ou Scala 2.9. Si vous êtes toujours sous Java 6, envisagez de passer à une version prise en charge.

Tout d'abord, créez un projet Maven et ajoutez la dépendance suivante dans votre pom:

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

Note : n'oubliez pas de mettre à jour le champ de version pour les dernières versions (maintenant> 0.10).

Le consommateur est initialisé à l'aide d'un objet `Properties` . Il y a beaucoup de propriétés vous permettant d'affiner le comportement du consommateur. Voici la configuration minimale requise:

```
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());
```

Le `bootstrap-servers` est une liste initiale de courtiers permettant au consommateur de découvrir le reste du cluster. Cela ne doit pas nécessairement être tous les serveurs du cluster: le client déterminera l'ensemble complet des courtiers actifs parmi les courtiers de cette liste.

Le `deserializer` indique au consommateur comment interpréter / désérialiser les clés et les valeurs du message. Ici, nous utilisons le `StringDeserializer` .

Enfin, le `group.id` correspond au groupe de consommateurs de ce client. Rappelez-vous: tous les consommateurs d'un groupe de consommateurs diviseront les messages entre eux (kafka agissant comme une file d'attente de messages), tandis que les consommateurs de différents groupes de consommateurs recevront les mêmes messages (kafka agissant comme un système de publication / abonnement).

D'autres propriétés utiles sont:

- `auto.offset.reset` : contrôle ce qu'il faut faire si le décalage stocké dans Zookeeper est manquant ou hors de portée. Les valeurs possibles sont les `latest` et les `earliest` . Tout le reste jettera une exception;

- `enable.auto.commit` : si `true` (valeur par défaut), le décalage du consommateur est périodiquement (voir `auto.commit.interval.ms`) enregistré en arrière-plan. Le paramétrer sur `false` et utiliser `auto.offset.reset=earliest` - permet de déterminer à partir de quel endroit le consommateur doit commencer au cas où aucune information de `auto.offset.reset=earliest` ne serait trouvée. signifie au `earliest` depuis le début de la partition de sujet assignée. `latest` moyens à partir du plus grand nombre de compensations validées disponibles pour la partition. Cependant, le consommateur Kafka reprendra toujours le dernier décalage validé tant qu'un enregistrement de décalage valide est trouvé (par exemple, en ignorant `auto.offset.reset` . Le meilleur exemple est lorsqu'un nouveau groupe de consommateurs s'abonne à un sujet. `auto.offset.reset` pour déterminer s'il faut commencer par le début (au plus tôt) ou la fin (la plus récente) du sujet.

- `session.timeout.ms` : un délai d'attente de session garantit que le verrou sera libéré en cas de panne du consommateur ou si une partition réseau isole le consommateur du coordinateur. Effectivement:

  > Lorsqu'il fait partie d'un groupe de consommateurs, chaque consommateur se voit attribuer un sous-ensemble de partitions à partir des sujets auxquels il est abonné. Ceci est essentiellement un verrou de groupe sur ces partitions. Tant que le verrou est maintenu, aucun autre membre du groupe ne pourra en lire. Lorsque votre consommateur est en bonne santé, c'est exactement ce que vous voulez. C'est la seule façon d'éviter la consommation en double. Mais si le consommateur meurt à la suite d'une défaillance de l'ordinateur ou de l'application, vous devez libérer ce verrou pour pouvoir affecter les partitions à un membre sain. la source

La liste complète des propriétés est disponible ici http://kafka.apache.org/090/documentation.html#newconsumerconfigs .

Création du consommateur et abonnement au sujet

Une fois que nous avons les propriétés, créer un consommateur est facile:

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

Après vous être abonné, le consommateur peut se coordonner avec le reste du groupe pour obtenir son affectation de partition. Tout cela est géré automatiquement lorsque vous commencez à consommer des données.

Sondage de base

Le consommateur doit pouvoir extraire des données en parallèle, potentiellement à partir de nombreuses partitions, sur de nombreux sujets susceptibles de s'étendre à de nombreux courtiers. Heureusement, tout cela est géré automatiquement lorsque vous commencez à consommer des données. Pour ce faire, il suffit d'appeler le `poll` en boucle et le consommateur gère le reste.

`poll` renvoie un ensemble (éventuellement vide) de messages provenant des partitions affectées.

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

Le code

Exemple de base

C'est le code le plus élémentaire que vous pouvez utiliser pour récupérer des messages à partir d'un sujet 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 );
            }
        }
    }
}
```

Exemple runnable

> Le consommateur est conçu pour être exécuté dans son propre thread. Il n'est pas sûr pour une utilisation multithread sans synchronisation externe et ce n'est probablement pas une bonne idée d'essayer.

Vous trouverez ci-dessous une simple tâche Runnable qui initialise le consommateur, s'abonne à une liste de rubriques et exécute la boucle d'interrogation indéfiniment jusqu'à son extinction externe.

```
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();
    }
}
```

Notez que nous utilisons un délai d'attente de `Long.MAX_VALUE` lors du sondage, il attendra donc indéfiniment un nouveau message. Pour fermer correctement le consommateur, il est important d'appeler sa méthode `shutdown()` avant de terminer l'application.

Un pilote pourrait l'utiliser comme ceci:

```
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)

Configuration et initialisation

Tout d'abord, créez un projet Maven et ajoutez la dépendance suivante dans votre pom:

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

Le producteur est initialisé à l'aide d'un objet `Properties` . Il y a beaucoup de propriétés vous permettant d'affiner le comportement du producteur. Voici la configuration minimale requise:

```
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");
```

Le `bootstrap-servers` est une liste initiale d'un ou plusieurs courtiers pour que le producteur puisse découvrir le reste du cluster. Les propriétés du `serializer` indiquent à Kafka comment la clé et la valeur du message doivent être codées. Ici, nous enverrons des messages de chaîne. Bien que cela ne soit pas obligatoire, la définition d'un `client.id` est toujours recommandée: cela vous permet de corréler facilement les requêtes sur le courtier avec l'instance cliente qui l'a `client.id` .

Les autres propriétés intéressantes sont:

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

Vous pouvez contrôler la *durabilité des messages* écrits sur Kafka via le paramètre `acks` . La valeur par défaut de «1» nécessite un accusé de réception explicite du leader de la partition indiquant que l'écriture a réussi. La garantie la plus forte fournie par Kafka est d' `acks=all` , ce qui garantit que le leader de la partition a non seulement accepté l'écriture, mais qu'il a été répliqué avec succès sur toutes les répliques in-sync. Vous pouvez également utiliser une valeur de «0» pour optimiser le débit, mais vous ne pouvez pas garantir que le message a bien été écrit dans le journal du courtier, car le courtier n'envoie même pas de réponse dans ce cas.

`retries` (default to> 0) détermine si le producteur essaie de renvoyer le message après un échec. Notez qu'avec les tentatives> 0, le réordonnancement des messages peut se produire car la nouvelle tentative peut se produire après une écriture suivante réussie.

Les producteurs de Kafka tentent de collecter les messages envoyés dans des lots pour améliorer le débit. Avec le client Java, vous pouvez utiliser `batch.size` pour contrôler la taille maximale en octets de chaque lot de messages. Pour donner plus de temps aux lots à remplir, vous pouvez utiliser `linger.ms` pour que le producteur `linger.ms` un envoi. Enfin, la compression peut être activée avec le paramètre `compression.type` .

Utilisez `buffer.memory` pour limiter la mémoire totale disponible au client Java pour la collecte des messages non envoyés. Lorsque cette limite est atteinte, le producteur bloque les envois supplémentaires aussi longtemps que `max.block.ms` avant de `max.block.ms` une exception. De plus, pour éviter de conserver des enregistrements indéfiniment en file d'attente, vous pouvez définir un délai d'attente à l'aide de `request.timeout.ms` .

La liste complète des propriétés est disponible ici . Je suggère de lire cet article de Confluent pour plus de détails.

Envoi de messages

La méthode `send()` est asynchrone. Lorsqu'il est appelé, il ajoute l'enregistrement à un tampon des enregistrements en attente et le retourne immédiatement. Cela permet au producteur de regrouper des enregistrements individuels pour des raisons d'efficacité.

Le résultat de l'envoi est un `RecordMetadata` spécifiant la partition à laquelle l'enregistrement a été envoyé et le décalage `RecordMetadata` il a été affecté. Étant donné que l'appel d'envoi est asynchrone, il renvoie un `Future` pour le RecordMetadata qui sera affecté à cet enregistrement. Pour consulter les métadonnées, vous pouvez soit appeler `get()` , qui bloquera jusqu'à la fin de la requête, soit utiliser un rappel.

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

Le code

```
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
    }
}
```

Lire Producteur / Consommateur en Java en ligne: https://riptutorial.com/fr/apache-kafka/topic/8974/producteur---consommateur-en-java

Chapitre 5: Sérialiseur / Désérialiseur personnalisé

Introduction

Kafka stocke et transporte des tableaux d'octets dans sa file d'attente. Les sérialiseurs (de) sont responsables de la traduction entre le tableau d'octets fourni par Kafka et les POJO.

Syntaxe

- public void configure (Map <String,?> config, boolean isKey);
- public T deserialize (Sujet de chaîne, octet [] octets);
- octet public [] sérialiser (sujet de chaîne, T obj);

Paramètres

paramètres	détails
config	les propriétés de configuration (`Properties`) transmises au `Producer` ou au `Consumer` lors de la création, sous forme de carte. Il contient des configurations standard de kafka, mais peut également être complété par une configuration définie par l'utilisateur. C'est le meilleur moyen de transmettre des arguments au sérialiseur (de).
C est la clé	Les sérialiseurs (de) personnalisés peuvent être utilisés pour les clés et / ou les valeurs. Ce paramètre vous indique lequel des deux cette instance va traiter.
sujet	le sujet du message actuel. Cela vous permet de définir une logique personnalisée en fonction du sujet source / destination.
octets	Le message brut à désérialiser
obj	Le message à sérialiser. Sa classe réelle dépend de votre sérialiseur.

Remarques

Avant la version 0.9.0.0, l'API Java Kafka utilisait les `Encoders` et les `Decoders` . Ils ont été remplacés par `Serializer` et `Deserializer` dans la nouvelle API.

Examples

Sérialiseur Gson (de)

Cet exemple utilise la bibliothèque gson pour mapper des objets Java sur des chaînes json. Les sérialiseurs (de) sont génériques, mais ils n'ont pas toujours besoin d'être!

Sérialiseur

Code

```
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
    }
}
```

Usage

Les sérialiseurs sont définis via les propriétés de production `key.serializer` et `value.serializer` .

Supposons que nous ayons une classe POJO nommée `SensorValue` et que nous voulons produire des messages sans aucune clé (les clés sont définies sur `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` est une configuration requise. Comme nous ne `key.serializer` pas les clés de message, nous conservons le `StringSerializer` livré avec kafka, qui peut gérer les `StringSerializer` `null`).

désérialiseur

Code

```
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() {}
}
```

Usage

Les désérialiseurs sont définis via les propriétés de consommateur `key.deserializer` et `value.deserializer` .

Supposons que nous ayons une classe POJO nommée `SensorValue` et que nous voulons produire des messages sans aucune clé (les clés sont définies sur `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 ...
}
```

Ici, nous ajoutons une propriété personnalisée à la configuration du consommateur, à savoir `CONFIG_VALUE_CLASS`. `GsonDeserializer` l'utilisera dans la méthode `configure()` pour déterminer la classe POJO à gérer (toutes les propriétés ajoutées aux `props` seront transmises à la méthode `configure` sous la forme d'une carte).

Lire Sérialiseur / Désérialiseur personnalisé en ligne: https://riptutorial.com/fr/apache-kafka/topic/8992/serialiseur---deserialiseur-personnalise

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec apache-kafka	Ali786, Community, Derlin, Laurel, Mandeep Lohan, Matthias J. Sax, Mincong Huang, NangSaigon, Vivek
2	Groupes de consommateurs et gestion des compensations	Matthias J. Sax, Sönke Liebau
3	les outils de la console kafka	Derlin
4	Producteur / Consommateur en Java	Derlin, ha9u63ar
5	Sérialiseur / Désérialiseur personnalisé	Derlin, G McNicol