消息队列 CKafka 版

开发指南

腾讯云

【联系我们】

我们致力于为您提供个性化的售前购买咨询服务，及相应的技术售后服务，任何问题请联系 4009100100或95716。

文档目录

开发指南
CKafka 事务管理

最近更新时间：2025-04-03 17:49:41

Kafka 的事务功能是为了支持在分布式环境中实现原子性操作而设计的。它允许生产者在发送消息时确保消息的完整性和一致性，特别是在需要多条消息作为一个整体进行处理的场景中。以下是 Kafka 事务的主要概念和功能介绍。

事务相关概念

事务的基本概念

原子性：事务中的所有操作要么全部成功，要么全部失败。Kafka 确保在事务中发送的消息要么被成功写入到主题中，要么不写入。
一致性：事务在执行前后，数据的状态应该保持一致。
隔离性：事务之间的操作是相互独立的，一个事务的执行不应影响其他事务的执行。
持久性：一旦事务被提交，其结果是永久性的，即使系统崩溃也不会丢失。

事务的工作流程

Kafka 的事务工作流程主要包括以下几个步骤：

1. **启动事务：**生产者在发送消息之前调用 `initTransactions()` 方法来初始化事务。
2. **发送消息：**生产者可以发送多条消息到一个或多个主题，这些消息会被标记为事务性消息。
3. **提交或中止事务：**
   - **提交事务：**如果所有消息都成功发送，生产者调用 `commitTransaction()` 方法来提交事务，所有消息将被写入到 Kafka。
   - **中止事务：**如果在发送过程中发生错误，生产者可以调用 `abortTransaction()` 方法来中止事务，所有消息将不会被写入。

事务的配置

要使用 Kafka 的事务功能，您需要在生产者配置中设置以下参数：

- `transactional.id`：每个事务性生产者都需要一个唯一的标识符。这个 ID 用于标识事务的所有消息。
- `acks`：设置为 all 以确保所有副本都确认消息。
- `enable.idempotence`：设置为 `true` 以启用幂等性，确保消息不会被重复发送。

事务的限制

- **性能开销：**使用事务会引入额外的性能开销，因为需要进行更多的协调和确认。

- **事务超时**：Kafka 对事务有超时限制，默认情况下为 60 秒。如果事务在此时间内未提交或中止，将会被自动中止。
- **消费者的处理**：消费者在处理事务性消息时需要注意，只有在事务提交后，消费者才能看到这些消息。

事务使用示例

producer

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

import java.util.Properties;

public class TransactionalProducerDemo {
    public static void main(String[] args) {
        // Kafka 配置
        Properties props = new Properties();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
"localhost:9092"); // Kafka broker 地址
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
"org.apache.kafka.common.serialization.StringSerializer");
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
"org.apache.kafka.common.serialization.StringSerializer");
        props.put(ProducerConfig.TRANSACTIONAL_ID_CONFIG, "my-
transactional-id"); // 事务 ID
        props.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, "true"); //
启用幂等性

        // 创建 Kafka 生产者
        KafkaProducer<String, String> producer = new KafkaProducer<>
(props);

        // 初始化事务
        producer.initTransactions();

        try {
            // 开始事务
            producer.beginTransaction();

            // 发送消息
            for (int i = 0; i < 10; i++) {
```

```
                ProducerRecord<String, String> record = new
ProducerRecord<>("my-topic", "key-" + i, "value-" + i);
                RecordMetadata metadata = producer.send(record).get();
// 发送消息并等待确认
                System.out.printf("Sent message: key=%s, value=%s,
partition=%d, offset=%d%n",
                                  record.key(), record.value(),
metadata.partition(), metadata.offset());
            }

            // 提交事务
            producer.commitTransaction();
            System.out.println("Transaction committed successfully.");
        } catch (Exception e) {
            // 如果发生异常，回滚事务
            producer.abortTransaction();
            System.err.println("Transaction aborted due to an error: " +
e.getMessage());
        } finally {
            // 关闭生产者
            producer.close();
        }
    }
}
```

consumer

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

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

public class TransactionalConsumerDemo {
    public static void main(String[] args) {
        // Kafka 配置
        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
"localhost:9092"); // Kafka broker 地址
```

```
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "my-consumer-group");
// 消费者组 ID
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
"org.apache.kafka.common.serialization.StringDeserializer");
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
"org.apache.kafka.common.serialization.StringDeserializer");
        props.put(ConsumerConfig.ISOLATION_LEVEL_CONFIG,
"read_committed"); // 只读取已提交的事务消息

        // 创建 Kafka 消费者
        KafkaConsumer<String, String> consumer = new KafkaConsumer<>
(props);

        // 订阅主题
        consumer.subscribe(Collections.singletonList("my-topic"));

        try {
            while (true) {
                // 拉取消息
                ConsumerRecords<String, String> records =
consumer.poll(Duration.ofMillis(100));
                for (ConsumerRecord<String, String> record : records) {
                    System.out.printf("Consumed message: key=%s,
value=%s, partition=%d, offset=%d%n",
                            record.key(), record.value(),
record.partition(), record.offset());
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            // 关闭消费者
            consumer.close();
        }
    }
}
```

Kafka 事务管理

在 Kafka 中，事务管理涉及到多个组件和数据结构，以确保事务的原子性和一致性。事务信息的内存占用主要与以下几个方面有关：

事务 ID 和 Producer ID

事务 ID：每个事务都有一个唯一的事务 ID，用于标识该事务。事务 ID 是由生产者在发送消息时指定的，通常是一个字符串。
Producer ID：每个生产者在连接到 Kafka 时会被分配一个唯一的 Producer ID。这个 ID 用于标识生产者的消息，并确保消息的顺序性和幂等性。

事务状态管理

Kafka 使用一个称为 事务状态日志 的内部主题来管理事务的状态。这个日志记录了每个事务的状态（如进行中、已提交、已中止）以及与该事务相关的消息。事务状态日志的管理涉及以下几个方面：

- **内存中的数据结构**：Kafka 在内存中维护一个数据结构（例如哈希表或映射），用于存储当前活动的事务信息。这些信息包括事务 ID、Producer ID、事务状态、时间戳等。
- **持久化存储**：事务状态日志会被持久化到磁盘，以确保在 Kafka 服务器重启或故障恢复时能够恢复事务状态。

事务信息的内存占用

事务信息的内存占用主要取决于以下两个因素：

- **活动事务的数量**：当前正在进行的事务数量直接影响内存占用。每个活动事务都会在内存中占用一定的空间。
- **事务的元数据**：每个事务的元数据（例如事务 ID、Producer ID、状态等）也会占用内存。具体的内存占用量取决于这些元数据的大小。

事务的清理

为了防止内存占用过高，Kafka 会根据配置的过期时间定期检查并清理已完成的事务，默认保留 **7 天**，过期删除。

事务常见的 FullGC / OOM 问题

从事务管理可以看出，事务信息会占用大量内存。其中影响事务信息占用内存大小的最直接的两个因素就是：事务 ID 的数量和 Producer ID 的数量。

- 其中事务 ID 的数量指的是客户端往 Broker 初始化、提交事务的数量，这个与客户端的事务新增提交频率强相关。
- Producer ID 指的是 Broker 内每个 Topic 分区存储的 producer 状态信息，因此 Producer ID 的数量与 broker 的分区数量强相关。

在事务场景中，事务 ID和 Producer ID 强绑定，如果同一个和事务 ID 绑定的 Producer ID 往 broker 内所有的分区都发送消息，那么一个 broker 内的 Producer ID 的数量理论上最多能达到事务 ID 数量与 broker 内分区数量的乘积。假设一个实例下的事务 ID 数量为 t，一个 broker 下的分区数量为 p，那么 Producer ID 的数量最大能达到 t * p。

> **说明：**
> 因此，假设一个 broker 下的事务 ID 数量为 t，平均事务内存占用大小为 tb，一个 broker 下的分区数量为 p，平均一个 Producer ID 占用大小为 pb，那么该 broker 内存中关于事务信息占用的内存大小为：t * tb + t * p * pb。

可以看出有两种场景可能会导致内存占用暴涨：

- 客户端频繁往实例初始化新增提交新的事务 ID。
- 同一个事务 ID 往多个分区发送数据，Producer ID 的叉乘数量会上涨的非常恐怖，很容易将内存打满。

> **说明：**
> 因此，无论是对 Flink 客户端还是自己实现的事务 producer，都尽量避免这两种场景。例如对于 Flink，可以适当降低 checkpoint 的频率，以减小由于事务 ID 前缀+随机串计算的事务 ID 变化的频率。另外就是尽量保证同一个事务 ID 往同一个分区发送数据。

Flink 使用事务注意事项

对于 Flink 有以下优化手段，来保证事务信息不会急剧膨胀：

- 客户端优化参数：Flink 加大 `checkpoint` 间隔（详情可参见 社区 ISSUE ）。
- Flink 生产任务可优化 `sink.partitioner` 为 `Fixed` 模式。

flink 参数说明：https://nightlies.apache.org/flink/flink-docs-master/zh/docs/connectors/table/kafka/

CKafka 版本选择建议

最近更新时间：2025-03-17 14:34:24

本文为您介绍腾讯云 CKafka 和社区版 Kafka 的兼容性，帮助您在使用腾讯云 CKafka 时根据业务需求选择更加适合您的版本。

概述

社区版 Kafka 目前共演进了 0.7.x 到 3.3.x 大概30个版本，从消息队列的角度可分为四个阶段：0.x、1.x、2.x、3.x。目前腾讯云针对这四个社区发展阶段均提供了云上兼容版本，基本覆盖了用户使用的主流 Kafka 版本。
其中 1.x 和 2.x 这两个大版本主要是对 Kafka Streams 的优化和改进，在消息引擎方面并未引入太多的重大功能特性（2.x 在事务特性方面有较大改进）。Kafka Streams 在 2.x 版本有较大改进，如果您是这些特性的用户，请至少选择 2.x 的版本。3.x 版本在 KRaft、mirrormaker2 等方面有较大改进，详情可以参考社区 release notes 。

兼容性说明

腾讯云 CKafka 兼容社区 Kafka，其中高版本和低版本是完全向下兼容的。例如：自建 0.10 版本的 Kafka，在云上选择0.10、1.1.1、2.4.1版本的 CKafka 均可；如果自建是高版本，不建议选择低版本（因为不确定业务是否使用高版本携带的特性）。
以下是兼容性说明：

CKafka 版本	可兼容社区版本	兼容性
0.10.2（已停售）	≤ 0.10.x	100%
1.1.1（已停售）	≤ 1.1.x	100%
2.4.1	≤ 2.4.x	100%
2.8.1（推荐）	≤ 2.8.x	100%
3.2.3	≤ 3.2.x	100%

关于 CKafka2.4.1 版本说明
CKafka 上线2.4版本时，社区的稳定分支为2.4.1版，后来社区有一个开发分支2.4.2版本，进行一些修复合入后，定位为2.4.2版本。后续社区删除了2.4.2版本，最终社区上没有2.4.2版本，因此 CKafka 之前显示的2.4.2版本和现在显示的2.4.1版本对齐。

CKafka 版本选择建议

- 如果是自建 Kafka 上云，建议选择对应的大版本即可。例如：自建 Kafka 是2.8.1版本，则选择 CKafka 的2.8.1版本。
- 当在云上找不到对应的版本时，建议向上选择版本。例如：自建是2.8.0版本，则建议使用2.8.1版本；自建是1.1.1版本，则建议使用版本2.4.1（因为 Broker 的每个版本是向下兼容的）。

CKafka 常见参数配置说明

最近更新时间：2025-02-10 17:37:32

Broker 配置参数说明

当前 CKafka broker 端的一些配置如下，供参考：

```
# 消息体的最大大小，单位是字节
message.max.bytes=1000012

# 是否允许自动创建 Topic，默认是 false，当前可以通过控制台或云 API 创建
auto.create.topics.enable=false

# 是否允许调用接口删除 Topic
delete.topic.enable=true

# Broker 允许的最大请求大小为16MB
socket.request.max.bytes=16777216

# 每个 IP 与 Broker 最多建立5000个连接
max.connections.per.ip=5000

# offset 保留时间，默认为7天
offsets.retention.minutes=10080

# 没有 ACL 设置时，允许任何人访问
allow.everyone.if.no.acl.found=true

# 日志分片大小为1GB
log.segment.bytes=1073741824

# 日志滚动检查间隔5分钟，当设置保留时间小于5分钟时，也可能需要等待5分钟才会清空日志
log.retention.check.interval.ms=300000
```

> **说明**
> 其他未列出的 Broker 配置参见 开源 Kafka 默认配置 。

Topic 配置参数说明

1. 选取合适的分区数量

从生产者的角度来看，向不同的 partition 写入是完全并行的；从消费者的角度来看，并发数完全取决于 partition 的数量（如果 consumer 数量大于 partition 数量，则必有 consumer 闲置）。因此选取合适的分区数量对于发挥 CKafka 实例的性能十分重要。
partition 的数量需要根据生产和消费的吞吐来判断。理想情况下，可以通过如下公式来判断分区的数目：

```
Num =  max( T/PT , T/CT ) = T / min( PT ,  CT )
```

其中，Num 代表 partition 数量，T 代表目标吞吐量，PT 代表生产者写入单个 partition 的最大吞吐，CT 代表消费者从单个 partition 消费的最大吞吐。则 partition 数量应该等于 T/PT 和 T/CT 中较大的那一个。
在实际情况中，生产者写入单个 partition 的最大吞吐 PT 的影响因素和批处理的规模、压缩算法、确认机制、副本数等有关。消费者从单个 partition 消费的最大吞吐 CT 的影响因素和业务逻辑有关，需要在不同场景下实测得出。
通常建议 partition 的数量一定要大于等于消费者的数量来实现最大并发。如果消费者数量是 5，则 partition 的数目也应该是 ⩾ 5 的。同时，过多的分区会导致生产吞吐的降低和选举耗时的增加，因此也不建议过多分区。提供如下信息供参考：

- 单个 partition 是可以实现消息的顺序写入的。
- 单个 partition 只能被同消费者组的单个消费者进程消费。
- 单个消费者进程可同时消费多个 partition，即 partition 限制了消费端的并发能力。
- partition 越多则失败后 leader 选举的耗时越长。
- offset 的粒度最细是在 partition 级别的，partition 越多，查询 offset 就越耗时。
- partition 的数量是可以动态增加的，只能增加不能减少。但增加会出现消息 rebalance 的情况。

2. 选取合适的副本

目前为了保证可用性副本数必须大于等于2，如果需要保障高可靠建议3副本。

> ⚠ **注意**
> 副本数会影响生产/消费流量，如3副本则实际流量 = 生产流量 × 3。

3. 日志保留时间

Topic 的 log.retention.ms 配置通过控制台实例的保留时间统一设置。

4. 其他 Topic 级别配置说明

```
# Topic 级别最大消息大小
max.message.bytes=1000012

# 0.10.2 版本消息格式为 V1 格式
message.format.version=0.10.2-IV0
```

```
# 不在 ISR 中的 replica 允许选择为 Leader，可用性高于可靠性，存在数据丢失风险。
unclean.leader.election.enable=true

# ISR 提交生产者请求的最小副本数。如果同步状态的副本数小于该值，服务器将不再接受。
request.required.acks为-1或all的写入请求。
min.insync.replicas=1
```

生产者配置指南

生产端常用参数配置如下，建议客户根据实际业务场景调整配置：

```
# 生产者会尝试将业务发送到相同的 Partition 的消息合包发送到 Broker，batch.size 设置合包的大小上限。默认为 16KB。batch.size 设太小会导致吞吐下降，设太大会导致内存使用过多。
batch.size=16384

# Kafka producer 的 ack 有 3 种机制，分别说明如下：
# -1 或 all：Broker 在 leader 收到数据并同步给所有 ISR 中的 follower 后，才应答给 Producer 继续发送下一条（批）消息。 这种配置提供了最高的数据可靠性，只要有一个已同步的副本存活就不会有消息丢失。注意：这种配置不能确保所有的副本读写入该数据才返回，可以配合 Topic 级别参数 min.insync.replicas 使用。
# 0：生产者不等待来自 broker 同步完成的确认，继续发送下一条（批）消息。这种配置生产性能最高，但数据可靠性最低（当服务器故障时可能会有数据丢失，如果 leader 已死但是 producer 不知情，则 broker 收不到消息）
# 1：生产者在 leader 已成功收到的数据并得到确认后再发送下一条（批）消息。这种配置是在生产吞吐和数据可靠性之间的权衡（如果leader已死但是尚未复制，则消息可能丢失）

# 用户不显式配置时，默认值为1。用户根据自己的业务情况进行设置
acks=1

# 控制生产请求在 Broker 等待副本同步满足 acks 设置的条件所等待的最大时间
timeout.ms=30000

# 配置生产者用来缓存消息等待发送到 Broker 的内存。用户要根据生产者所在进程的内存总大小调节
buffer.memory=33554432

# 当生产消息的速度比 Sender 线程发送到 Broker 速度快，导致 buffer.memory 配置的内存用完时会阻塞生产者 send 操作，该参数设置最大的阻塞时间
max.block.ms=60000

# 设置消息延迟发送的时间(ms)，这样可以等待更多的消息组成 batch 发送。默认为0表示立即发送。当待发送的消息达到 batch.size 设置的大小时，不管是否达到 linger.ms 设置的时
```

```
间，请求也会立即发送
# 推荐用户根据实际使用场景，设置linger.ms在100~1000之间，更大的取值相对有更大的吞吐但会相应增加时延
linger.ms=100

# 设置分区缓存消息量(bytes)，达到该数值时生产者将批量的消息发送给broker。默认值为16384，过小的batch.size会增加发送请求次数可能会损耗性能和影响稳定性，用户根据实际场景，可适当增大该值。注：该值为上限值，当还未达到时若时间已经达到linger.ms时生产者会发送消息
batch.size=16384

# 生产者能够发送的请求包大小上限，默认为1MB。在修改该值时注意不能超过 Broker 配置的包大小上限16MB
max.request.size=1048576

# 压缩格式配置，目前 0.9(包含)以下版本不允许使用压缩，0.10（包含）以上不允许使用 GZip 压缩
compression.type=[none, snappy, lz4]

# 客户端发送给 Broker 的请求的超时时间，不能小于 Broker 配置的 replica.lag.time.max.ms，目前该值为10000ms
request.timeout.ms=30000

# 客户端在每个连接上最多可发送的最大的未确认请求数，该参数大于1且 retries 大于0时可能导致数据乱序。 希望消息严格有序时，建议客户将该值设置1
max.in.flight.requests.per.connection=5


# 请求发生错误时重试次数，建议将该值设置为大于0，失败重试最大程度保证消息不丢失
retries=0

# 发送请求失败时到下一次重试请求之间的时间
retry.backoff.ms=100
```

消费者配置指南

消费端常用参数配置如下，建议客户根据实际业务场景调整配置：

```
# 是否在消费消息后将 offset 同步到 Broker，当 Consumer 失败后就能从 Broker 获取最新的 offset
enable.auto.commit=true

# 当 auto.commit.enable=true 时，自动提交 Offset 的时间间隔，建议设置至少1000
```

```
auto.commit.interval.ms=5000

# 当 Broker 端没有 offset（如第一次消费或 offset 超过7天过期）时如何初始化
offset，当收到 OFFSET_OUT_OF_RANGE 错误时，如何重置 Offset
# earliest：表示自动重置到 partition 的最小 offset
# latest：默认为 latest，表示自动重置到 partition 的最大 offset
# none：不自动进行 offset 重置，抛出 OffsetOutOfRangeException 异常
auto.offset.reset=latest

# 标识消费者所属的消费分组
group.id=""

# 使用 Kafka 消费分组机制时，消费者超时时间。当 Broker 在该时间内没有收到消费者的心
跳时，认为该消费者故障失败，Broker 发起重新 Rebalance 过程。目前该值的配置必须在
Broker 配置group.min.session.timeout.ms=6000和
group.max.session.timeout.ms=300000 之间
session.timeout.ms=10000

# 使用 Kafka 消费分组机制时，消费者发送心跳的间隔。这个值必须小于
session.timeout.ms，一般小于它的三分之一
heartbeat.interval.ms=3000

# 使用 Kafka 消费分组机制时，再次调用 poll 允许的最大间隔。如果在该时间内没有再次调
用 poll，则认为该消费者已经失败，Broker 会重新发起 Rebalance 把分配给它的
partition 分配给其他消费者
max.poll.interval.ms=300000

# Fecth 请求最少返回的数据大小。默认设置为 1B，表示请求能够尽快返回。增大该值会增加吞
吐，同时也会增加延迟
fetch.min.bytes=1

# Fetch 请求最多返回的数据大小，默认设置为 50MB
fetch.max.bytes=52428800

# Fetch 请求等待时间
fetch.max.wait.ms=500

# Fetch 请求每个 partition 返回的最大数据大小，默认为1MB
max.partition.fetch.bytes=1048576

# 在一次 poll 调用中返回的记录数
max.poll.records=500
```

```
# 客户端请求超时时间，如果超过这个时间没有收到应答，则请求超时失败
request.timeout.ms=305000
```

CKafka 数据压缩

最近更新时间：2025-03-17 14:34:24

操作场景

数据压缩可以减少网络 IO 传输量，减少磁盘存储空间。您可以通过本文档，了解数据压缩支持的消息格式，并根据需求配置数据压缩。

消息格式

目前 CKafka 支持两类消息格式，分别为V1版本和V2版本（在0.11.0.0引入）。目前 CKafka 兼容0.9、0.10、1.1、2.4、2.8和3.2版本。
不同版本对应不同的配置，说明如下：

- 消息格式转换主要是为了兼容老版本的消费者程序，在一个 CKafka 集群中通常同时保存多种版本的消息格式（V1/V2）。
- Broker 端会对新版本消息执行向老版本格式的转换，该过程中会涉及消息的解压缩和重新压缩。
- 消息格式转换对性能的影响很大，除了增加额外的压缩和解压缩操作之外，还会让 CKafka 丧失其优秀的 零拷贝（Zero-copy）特性。因此，**一定要保证消息格式的统一**。
- 零拷贝（Zero-copy）：数据在磁盘和网络进行传输时，避免昂贵的内核态数据拷贝，从而实现快速的数据传输。

压缩算法对比

官方推荐使用的压缩算法为 Snappy 算法。分析过程如下：
评估一个压缩算法的优劣，主要有两个指标：压缩比、压缩/解压缩吞吐量。
CKafka 2.1.0之前的版本支持三种压缩算法：GZIP、Snappy、LZ4。
在 CKafka 的实际使用中，三种算法的**性能指标**对比如下：

- 压缩比：LZ4 > GZIP > Snappy
- 吞吐量：LZ4 > Snappy > GZIP

物理资源占用如下：

- 带宽：由于 Snappy 的压缩比最低，因此占用的网络带宽最大。
- CPU：在压缩时 Snappy 使用更多的 CPU，在解压缩时 GZIP 使用更多的 CPU。

因此，正常情况下三种压缩算法的推荐排序为：LZ4 > GZIP > Snappy。
经过长时间的现网运行试验，发现在大多数情况下上面的模型是没问题的。但是在某些极端情况下 LZ4 压缩算法会导致 CPU 负载增大。
经分析是业务的源数据内容不一样，导致压缩算法的性能表现不一样。 故建议对 CPU 指标敏感的用户采用更为稳定的 Snappy 压缩算法。

> **说明：**

> CKafka 不推荐使用 Gzip 压缩算法。开启 Gzip 压缩对 CKafka 服务端有额外的 CPU 消耗。根据压测数据，如果开启 Gzip 压缩，建议预留 75% 左右的带宽 buffer（预留比例仅供参考，实际使用时需要观测具体的监控数据判断）。
> 例如：带宽 40MB/s 的实例，开启 Gzip 压缩后，建议将带宽提升到 40/（1-75%）= 160MB/s。

配置数据压缩

生产者可通过下述方法配置数据压缩：

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("acks", "all");
props.put("key.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
props.put("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer");

// Producer 启动后，生产的每个消息集合都会经过压缩，能够很好地节省网络传输带宽和
Kafka Broker 端的磁盘占用

// 请注意不同版本对应不同的配置，0.9及以下版本不允许使用压缩。1.1及以下版本默认不支持
Gzip 压缩格式。

props.put("compression.type", " lz4 ");
Producer<String, String> producer = new KafkaProducer<>(props);

大部分情况下，Broker 从 Producer 接收到消息后，仅仅只是原封不动地保存，而不会对其进
行任何修改
```

说明与注意

- 发送数据到 CKafka，不能设置压缩 compression.codec。
- 1.1及以下版本默认不支持 Gzip 压缩格式，如果需要支持，请 提交工单 申请。
  Gzip 压缩对于 CPU 的消耗较高，使用 Gzip 会导致所有的消息都是 InValid 消息。**CKafka 不推荐使用 Gzip压缩。**
  开启 Gzip 后 CPU 占用率高，成为带宽使用瓶颈。若开启 Gzip，推荐调大生产端的linger.ms、batch.size 配置。
- 使用 LZ4 压缩方法时，程序不能正常运行，可能的原因是：消息格式错误。请您检查 CKafka 版本，确认适用的消息格式是否正确。
- 不同 CKafka Client 的 SDK 设置方式不同，您可以通过开源社区进行查询（例如 C/C++ Client 的说明），设置消息格式的版本。

接入低版本自建 Kafka

最近更新时间：2024-10-10 17:36:02

CKafka 兼容0.9及以上的生产/消费接口（目前可以直接购买的版本包括 2.4.1、2.8.1、3.2.3 版本），如果接入低版本（例如0.8版本）的自建 Kafka，您需要对接口进行相应改造。本文将从生产端和消费端对比0.8版本 Kafka 和高版本 Kafka，并提供改造方式。

Kafka Producer

概述

Kafka 0.8.1版本中，Producer API 被重写。该客户端为官方推荐版本，其拥有更好的性能和更多的功能，社区将维护新版本的 Producer API。

新旧版本 Producer API 对比

- 新版 Producer API Demo

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:4242");
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");
Producer<String, String> producer = new KafkaProducer<>(props);
producer.send(new ProducerRecord<String, String>("my-topic",
Integer.toString(0), Integer.toString(0)));
producer.close();
```

- 旧版 Producer API Demo

```
Properties props = new Properties();
props.put("metadata.broker.list", "broker1:9092");
props.put("serializer.class", "kafka.serializer.StringEncoder");
props.put("partitioner.class", "example.producer.SimplePartitioner");
props.put("request.required.acks", "1");
ProducerConfig config = new ProducerConfig(props);
```

```
Producer<String, String> producer = new Producer<String, String>
(config);
KeyedMessage<String, String> data = new KeyedMessage<String, String>
("page_visits", ip, msg);
producer.send(data);
producer.close();
```

可以看出新旧版本的使用方法基本一致，只有一些参数的配置不同，改造代价不大。

兼容性说明

对于 Kafka 而言，0.8.x 版本的 Producer API 都可以顺利接入 CKafka，无需改造。推荐使用新版 Kafka Producer API。

Kafka Consumer

概述

开源 Apache Kafka 0.8版本中提供了两种消费者 API，分别为：

- High Level Consumer API（屏蔽配置细节）
- Simple Consumer API（配置细节支持指定）

Kafka 0.9.x 版本引入了 New Consumer，其融合了 Old Consumer（0.8版本）两种 Consumer API 的特性，减轻了 ZooKeeper 的负载。
因此下文给出了0.8版本 Consumer 转换为0.9版本 New Consumer 的方式。

新旧版本 Consumer API 对比

0.8版本 Consumer API

- **High Level Consumer API**（参见 Demo）
  如果您只需要数据而不考虑消息 offset 相关的处理时，High Level API 可以满足一般性消费要求。High Level Consumer API 围绕着 Consumer Group 逻辑概念展开，屏蔽 Offset 管理、具有 Broker 异常处理、Consumer 负载均衡功能。使开发者可以快速上手 Consumer 客户端。
  在使用 High Level Consumer 时需要注意以下几点：
- 如果消费线程大于 Partition 个数，某些消费线程将无法获得数据。
- 如果 Partition 个数大于线程数目，某些线程会消费多个 Partition。
- Partition 和消费者变动会影响 Rebalance。
- **Low Level Consumer API**（参见 Demo）
  如果使用者关心消息的 offset 并且希望进行重复消费或者跳读等功能、又或者希望指定某些 partition 进行消费时和确保更多消费语义时推荐使用 Low Level Consumer API。但是使用者需要自己处理 Offset 以及 Broker 的异常情况。
  在使用 Low Level Consumer 时需要注意以下几点：

- 自行跟踪维护 Offset，控制消费进度。
- 查找 Topic 相应 Partition 的 Leader，以及处理 Partition 变更情况。

0.9版本 New Consumer API

Kafka 0.9.x 版本引入了 New Consumer，其融合了 Old Consumer 两种 Consumer API 的特性，同时提供消费者的协调(高级 API)和 lower-level 访问，并构建自定义的消费策略。New Consumer 还简化了消费者客户端，引入中心 Coordinator，解决分别连接 ZooKeeper 产生的 Herd Effect 和 Split Brain 问题，同时也减轻了 ZooKeeper 的负载。

优势：

- Coordinator 引入
  当前版本的 High Level Consumer 存在 Herd Effect 和 Split Brain 的问题。将失败探测和 Rebalance 的逻辑放到一个高可用的中心 Coordinator，那么这两个问题即可解决。同时还可很大程度的减少 ZooKeeper 的负载。
- 允许自己分配 Partition
  为了保持本地每个分区的一些状态不变，所以需要将 Partition 的映射也保持不变。另外一些场景是为了让 Consumer 与地域相关的 Broker 关联。
- 允许自己管理 Offset
  可以根据自己需要去管理 Offset，实现重复、跳跃消费等语意。
- Rebalance 后触发用户指定的回调
- 非阻塞式 Consumer API

新旧版本 Consumer API 功能对比

种类	引入版本	Offset 自动保存	Offset 自行管理	自动进行异常处理	Rebalance 自动处理	Leader 自动查找	优缺点
High Level Consumer	Before 0.9	支持	不支持	支持	支持	支持	Herd Effect 和 Split Brain
Simple Consumer	Before 0.9	不支持	支持	不支持	不支持	不支持	需要处理多种异常情况
New Consumer	After 0.9	支持	支持	支持	支持	支持	成熟，当前版本推荐

Old Consumer 转换 New Consumer

- New Consumer

```
//config中主要变化是 ZooKeeper 参数被替换了
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("group.id", "test");
props.put("enable.auto.commit", "true");
props.put("auto.commit.interval.ms", "1000");
props.put("session.timeout.ms", "30000");
props.put("key.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
// 相比old consumer 而言，这里创建消费者更加简单
KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
consumer.subscribe(Arrays.asList("foo", "bar"));
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records)
       System.out.printf("offset = %d, key = %s, value = %s",
record.offset(), record.key(), record.value());}
```

- Old Consumer（High Level）

```
//  old consumer 需要 ZooKeeper
Properties props = new Properties();
props.put("zookeeper.connect", "localhost:2181");
props.put("group.id", "test");
props.put("auto.commit.enable", "true");
props.put("auto.commit.interval.ms", "1000");
props.put("auto.offset.reset", "smallest");
ConsumerConfig config = new ConsumerConfig(props);
// 需要创建connector
ConsumerConnector connector =
Consumer.createJavaConsumerConnector(config);
// 创建message stream
Map<String, Integer> topicCountMap = new HashMap<String, Integer>();
topicCountMap.put("foo", 1);
Map<String, List<KafkaStream<byte[], byte[]>>> streams =
 connector.createMessageStreams(topicCountMap);
// 获取数据
KafkaStream<byte[], byte[]> stream = streams.get("foo").get(0);
```

```
ConsumerIterator<byte[], byte[]> iterator = stream.iterator();
MessageAndMetadata<byte[], byte[]> msg = null;
while (iterator.hasNext()) {
      msg = iterator.next();
      System.out.println(//
              " group " + props.get("group.id") + //
              ", partition " + msg.partition() + ", " + //
               new String(msg.message()));
}
```

可以看到，改造成 New Consumer 编写更加简单，最主要的变化是将 ZooKeeper 参数的输入替代成了 Kafka 地址输入。同时，New Consumer 也增加了与 Coodinator 交互的参数配置，一般情况下使用默认配置就足够。

兼容性说明

CKafka 与开源社区高版本的 Kafka 一致，支持重写后的 New Consumer API，屏蔽了 Consumer 客户端与 Zookeeper 的交互（Zookeeper不再向用户暴露）。New Consumer 解决原有与 Zookeeper 直接交互的 Herd Effect 和 Split Brain 问题，以及融合了原有 Old Consumer 的特性，使消费环节更加可靠。