消息队列 CKafka 版

实践教程

产品文档

腾讯云

文档目录

实践教程
CKafka 客户端实践教程
生产消费实践教程

最近更新时间：2024-07-04 16:00:59

本文主要介绍消息队列 CKafka 版生产和消费消息的实践教程，帮助您降低消费消息出错的可能性。

生产消息

Topic 使用推荐

配置要求：**推荐节点的整倍数副本，减少数据倾斜问题，同步复制最小同步副本数为2**，且同步副本数不能等于 Topic 副本数，否则宕机1个副本会导致无法生产消息。
创建方式：支持选择是否开启 CKafka 自动创建 Topic 的开关。选择开启后，表示生产或消费一个未创建的 Topic 时，会自动创建一个包含3个分区和2个副本的 Topic。

分区数估计

为实现尽可能的数据均衡，分区数建议为节点数的整倍数，同时在知道预估流量的基础上，按照1MB/s一个分区设置分区数，如100MB的 Topic 吞吐，建议设置分区数为100。

失败重试

分布式环境下，由于网络等原因，消息偶尔会出现发送失败的情况，其原因可能是消息已经发送成功但是 ACK 机制失败或者是消息确实没有发送成功。
您可以根据业务需求，设置以下重试参数：

参数	说明
retries	重试次数，默认值为3，就对于数据丢失零容忍的应用而言，请考虑设置为：Integer.MAX_VALUE（有效且最大）。
retry.backoff.ms	重试间隔，建议设置为1000。

这样将能够应对 Broker 的 Leader 分区出现无法立刻响应 Producer 请求的情况。

异步发送

发送接口是异步的，如果您想接收发送的结果，可用使用 send 方法中的Callback接口进行获取发送结果。

一个 Producer 对应一个应用

Producer 是线程安全的，且可以往任何 Topic 发送消息。通常情况下，建议一个应用对应一个 Producer。

Acks

Kafka 的 ACK 机制，指 Producer 的消息发送确认机制，在 Kafka 的 0.10.x 版本上，其设置值是 Acks，而在 0.8.x 版本上，则为 request.required.acks，Acks 的设置将直接影响到 Kafka 集群的吞吐量和消息可靠性。
Acks 的参数说明如下：

参数	说明
acks=0	无需服务端的 Response。性能较高、丢数据风险较大。
acks=1	服务端主节点写成功即返回 Response。性能中等、丢数据风险中等、主节点宕机可能导致数据丢失。
acks=all	服务端主节点写成功且 ISR 中的节点同步成功才返回 Response。性能较差、数据较为安全、主节点和备节点都宕机才会导致数据丢失。

一般建议选择 acks=1，重要的服务可以设置 acks=all。

Batch

一般情况下，消息队列 CKafka 的 Topic 会有多个分区，Producer 客户端在向服务端发送消息时，需要先确认往哪个 Topic 的哪个分区发送。在给同一个分区发送多条消息时，Producer 客户端会将相关消息打包成一个 Batch，批量发送到服务端。Producer 客户端在处理 Batch 时，是有额外开销的。一般情况下，小 Batch 会导致 Producer 客户端产生大量请求，造成请求队列在客户端和服务端的排队，并造成相关机器的 CPU 升高，从而整体推高了消息发送和消费延迟。一个合适的 Batch 大小，可以减少发送消息时客户端向服务端发起的请求次数，在整体上提高消息发送的吞吐和延迟。
Batch 参数说明如下：

参数	说明
batch.size	发往每个分区（Partition）的消息缓存量（消息内容的字节数之和，不是条数）。达到设置的数值时，就会触发一次网络请求，然后 Producer 客户端把消息批量发往服务器。
linger.ms	每条消息在缓存中的最长时间。若超过这个时间，Producer 客户端就会忽略 batch.size 的限制，立即把消息发往服务器。
buffer.memory	所有缓存消息的总体大小超过这个数值后，就会触发把消息发往服务器，此时会忽略 batch.size 和 linger.ms 的限制。buffer.memory 的默认数值是 32MB，对于单个 Producer 而言，可以保证足够的性能。

说明：

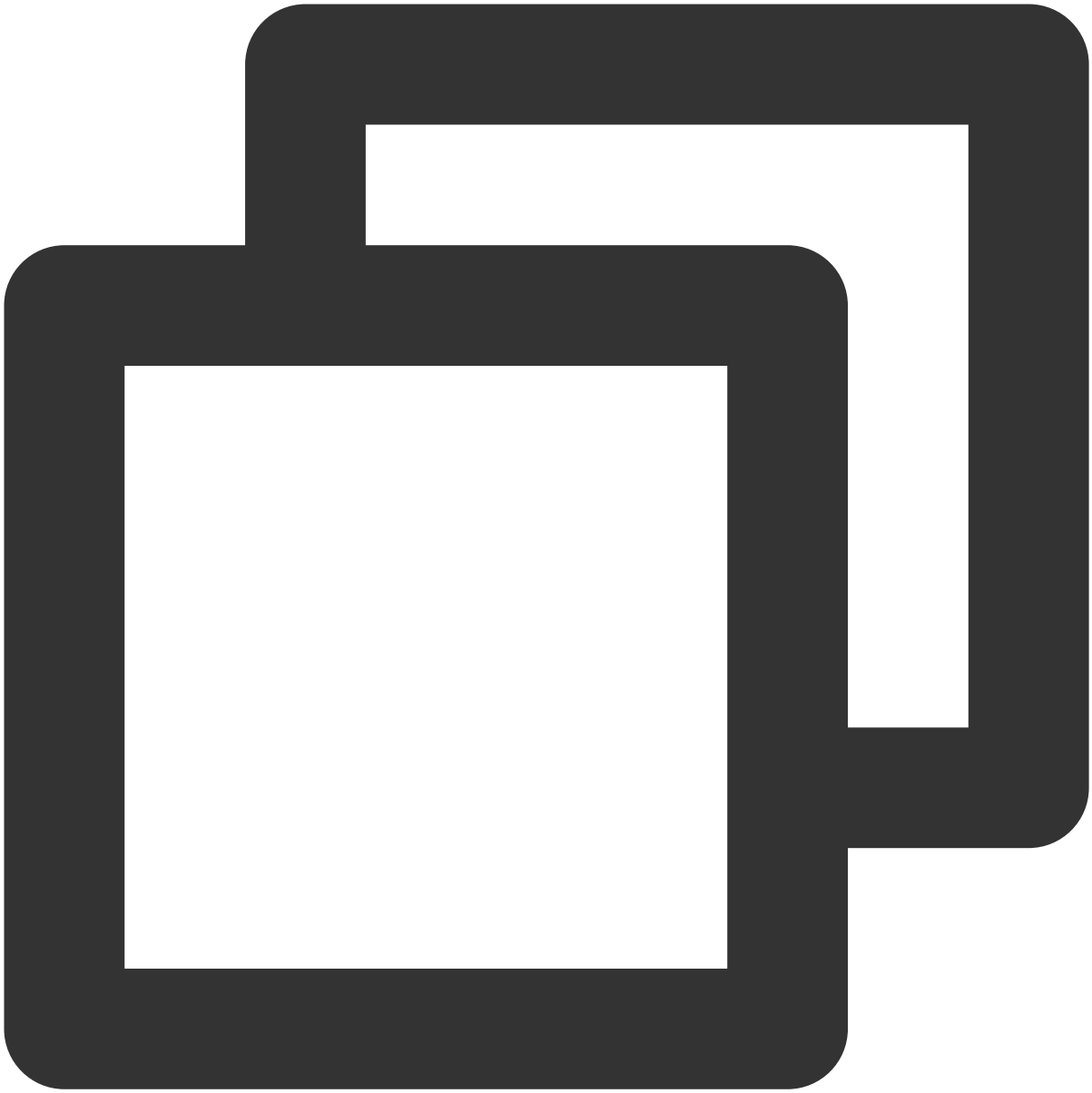
如果您在同一个 JVM 中启动多个 Producer，那么每个 Producer 都有可能占用 32MB 缓存空间，此时便有可能触发 OOM（Out of Memory），此时您需要考虑 buffer.memory 的大小，避免触发 OOM。
您可以根据具体业务需求进行参数设置值的调整。Producer 客户端什么时候把消息批量发送至服务器是由 batch.size 和 linger.ms 共同决定的。您可以根据具体业务需求进行调整。为了提升发送的性能，保障服务的稳定性， 建议您设置 batch.size=16384 和 linger.ms=1000。

Key 和 Value

消息队列 CKafka 的消息有 Key（消息标识）和 Value（消息内容）两个字段。
为了便于追踪，请为消息设置一个唯一的 Key。您可以通过 Key 追踪某消息，打印发送日志和消费日志，了解该消息的生产和消费情况。
如果消息发送量较大，建议不要设置 Key，并使用黏性分区策略。

黏性分区

只有发送到相同分区的消息，才会被放到同一个 Batch 中，因此决定一个 Batch 如何形成的一个因素是消息队列 Kafka Producer 端设置的分区策略。消息队列 Kafka Producer 允许通过设置 Partitioner 的实现类来选择适合自己业务的分区。在消息指定 Key 的情况下，消息队列 Kafka Producer 的默认策略是对消息的Key进行哈希，然后根据哈希结果选择分区，保证相同 Key 的消息会发送到同一个分区。
在消息没有指定 Key 的情况下，消息队列 Kafka 2.4版本之前的默认策略是循环使用主题的所有分区，将消息以轮询的方式发送到每一个分区上。但是，这种默认策略 Batch 的效果会比较差，在实际使用中，可能会产生大量的小 Batch，从而使得实际的延迟增加。鉴于该默认策略对无 Key 消息的分区效率低问题，消息队列 Kafka 在2.4版本引入了黏性分区策略（Sticky Partitioning Strategy）。
黏性分区策略主要解决无 Key 消息分散到不同分区，造成小 Batch问题。其主要策略是如果一个分区的 Batch 完成后，就随机选择另一个分区，然后后续的消息尽可能地使用该分区。这种策略在短时间内看，会将消息发送到同一个分区，如果拉长整个运行时间，消息还是可以均匀地发布到各个分区上的。这样可以避免消息出现分区倾斜，同时还可以降低延迟，提升服务整体性能。
如果您使用的消息队列 Kafka Producer 客户端是2.4及以上版本，默认的分区策略就采用黏性分区策略。如果您使用的 Producer 客户端版本小于2.4，可以根据黏性分区策略原理，自行实现分区策略，然后通过参数 partitioner.class 设置指定的分区策略。
关于黏性分区策略实现，您可以参见如下 Java 版代码实现。该代码的实现逻辑主要是根据一定的时间间隔，切换一次分区。



```
public class MyStickyPartitioner implements Partitioner {

    // 记录上一次切换分区时间。
    private long lastPartitionChangeTimeMillis = 0L;
    // 记录当前分区。
    private int currentPartition = -1;
    // 分区切换时间间隔，可以根据实际业务选择切换分区的时间间隔。
    private long partitionChangeTimeGap = 100L;

    public void configure(Map<String, ?> configs) {}
```

```
/**
 * Compute the partition for the given record.
 *
 * @param topic The topic name
 * @param key The key to partition on (or null if no key)
 * @param keyBytes serialized key to partition on (or null if no key)
 * @param value The value to partition on or null
 * @param valueBytes serialized value to partition on or null
 * @param cluster The current cluster metadata
 */
public int partition(String topic, Object key, byte[] keyBytes, Object value, b

    // 获取所有分区信息。
    List<PartitionInfo> partitions = cluster.partitionsForTopic(topic);
    int numPartitions = partitions.size();

    if (keyBytes == null) {
        List<PartitionInfo> availablePartitions = cluster.availablePartitionsFo
        int availablePartitionSize = availablePartitions.size();

        // 判断当前可用分区。
        if (availablePartitionSize > 0) {
            handlePartitionChange(availablePartitionSize);
            return availablePartitions.get(currentPartition).partition();
        } else {
            handlePartitionChange(numPartitions);
            return currentPartition;
        }
    } else {
        // 对于有key的消息，根据key的哈希值选择分区。
        return Utils.toPositive(Utils.murmur2(keyBytes)) % numPartitions;
    }
}

private void handlePartitionChange(int partitionNum) {
    long currentTimeMillis = System.currentTimeMillis();

    // 如果超过分区切换时间间隔，则切换下一个分区，否则还是选择之前的分区。
    if (currentTimeMillis - lastPartitionChangeTimeMillis >= partitionChangeTim
        || currentPartition < 0 || currentPartition >= partitionNum) {
        lastPartitionChangeTimeMillis = currentTimeMillis;
        currentPartition = Utils.toPositive(ThreadLocalRandom.current().nextInt
    }
}

public void close() {}
```

```
}
```

分区顺序

单个分区（Partition）内，消息是按照发送顺序储存的，是基本有序的。每个主题下面都有若干分区，如果消息被分配到不同的分区中，不同 Partition 之间不能保证顺序。
如果需要进行消息具有消费顺序性，可以在生产端指定这一类消息的 key，这类消息都用相同的 key 进行消息发送，CKafka 就会根据 key 哈希取模选取其中一个分区进行存储，由于一个分区只能由一个消费者进行监听消费，此时消息就具有消息消费的顺序性了。
数据倾斜
Kafka Broker数据倾斜问题通常是由于分区分布不均匀或者生产者发送数据的key分布不均匀导致的，会引发几类问题：
1. 整体流量没有限流，但是节点局部限流；
2. 某些节点负载过快，导致整体kafka使用率不高，影响整体吞吐。
针对该类问题可以通过以下方式进行优化：
3. 使用合理分区数，分区数保障为节点数的整倍数。
4. 合理的分区策略，例如：RoundRobin（轮询）、Range（范围）和Sticky（粘性）或者自定义的分区策略，均衡发送消息。
5. 查是否使用Key进行发送，如果使用了Key进行发送，尽量设计策略让Key更加分区均衡。

消费消息

消费消息基本流程

1. Poll 数据。
2. 执行消费逻辑。
3. 再次 poll 数据。

负载均衡

每个 Consumer Group 可以包含多个 Consumer，并将参数 group.id 设置成相同的值，属于同一个 Consumer Group 的 Consumer 会负责消费订阅的 Topic。
例如：Consumer Group A 订阅了 Topic A，并开启三个消费实例 C1、C2、C3，则发送到 Topic A 的每条消息最终只会传给 C1、C2、C3 的某一个。CKafka 默认会均匀地把消息传给各个消费实例，以做到消费负载均衡。
CKafka 负载均衡的内部原理是：把订阅的 Topic 的分区，平均分配给各个 Consumer。因此，Consumer 的个数不要大于分区的数量，否则会有消费实例分配不到任何分区而处于空跑状态，尽量保证消费者数量能被分区总数整除。除了第一次启动上线之外，后续消费实例发生重启、增加、减少，分区数发生增加等变更时，都会触发一次重均衡。

频繁出现 Rebalance

如果频繁出现 Rebalance，可能有以下几种可能：

1. 消费者消费处理耗时很长

2. 消费某一个异常消息导致消费者阻塞或者失败

3. 心跳超时会引发 Rebalance

4. v0.10.2之前版本的客户端：Consumer 没有独立线程维持心跳，而是把心跳维持与 poll 接口耦合在一起。其结果就是，如果用户消费出现卡顿，就会导致Consumer 心跳超时，引发 Rebalance。v0.10.2及之后版本的客户端：如果消费时间过慢，超过一定时间（max.poll.interval.ms设置的值，默认5分钟）未进行 poll 拉取消息，则会导致客户端主动离开队列，而引发 Rebalance。

可以通过优化消费处理提高消费速度和参数调整等方法解决：

1. 消费端需要和 Broker 版本保持一致。

2. 参考以下说明调整参数值：

session.timeout.ms：v0.10.2之前的版本可适当提高该参数值，需要大于消费一批数据的时间，但不要超过30s，建议设置为25s；而v0.10.2及其之后的版本，保持默认值10s即可。

max.poll.records：降低该参数值，建议远远小于<单个线程每秒消费的条数> <消费线程的个数> <max.poll.interval.ms>的积。

max.poll.interval.ms：该值要大于<max.poll.records> / (<单个线程每秒消费的条数> * <消费线程的个数>)的值。

3. 尽量提高客户端的消费速度，消费逻辑另起线程进行处理，针对耗时进行监控。

4. 减少 Group 订阅 Topic 的数量，一个 Group 订阅的 Topic 最好不要超过5个，建议一个 Group 只订阅一个 Topic。

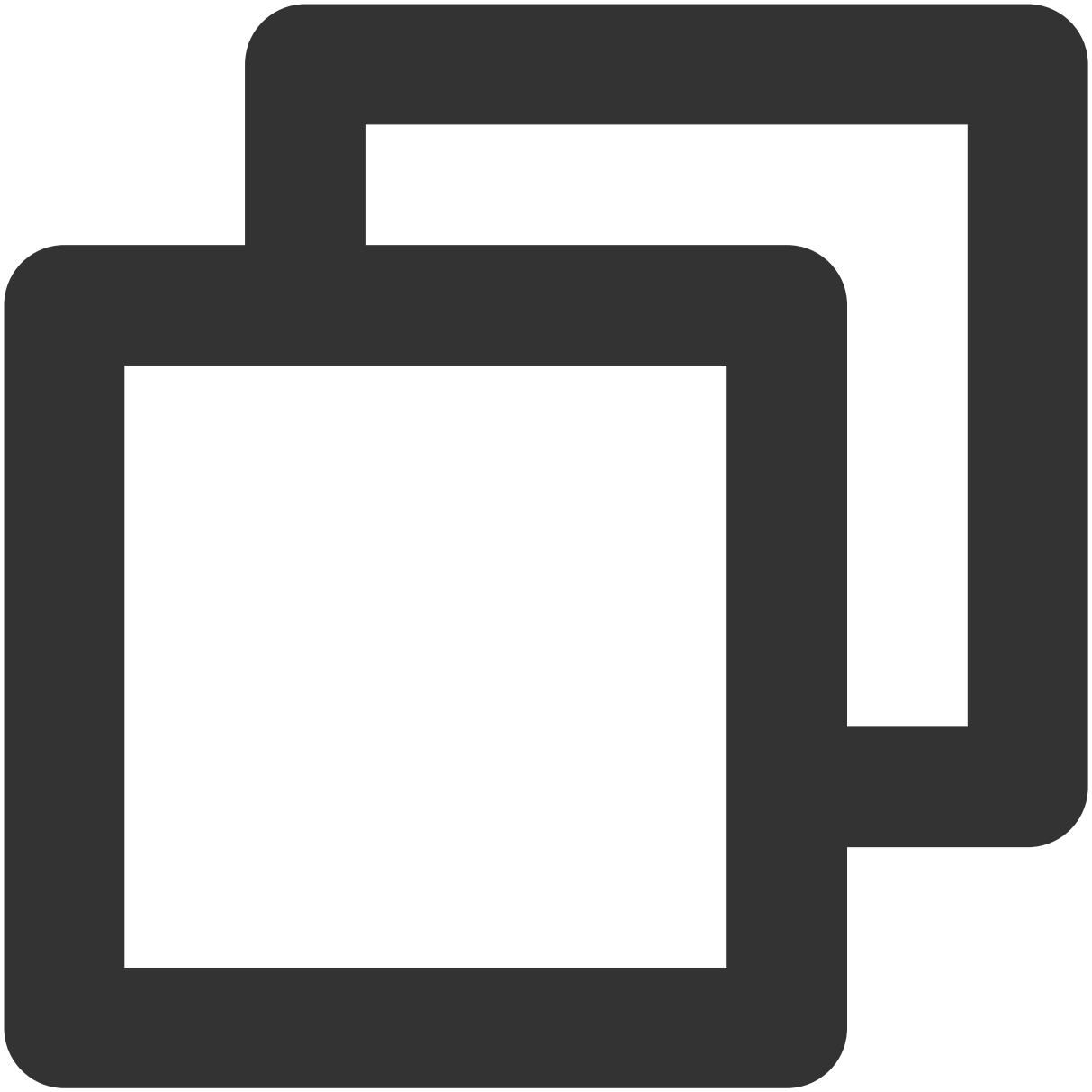
订阅关系

同一个 Consumer Group 内，建议各个消费实例订阅的 Topic 保持一致，避免给排查问题带来干扰。

Consumer Group 订阅多个 Topic

一个 Consumer Group 可以订阅多个Topic，多个 Topic 的消息被 Cosumer Group 中的 Consumer 均匀消费。例如 Consumer Group A 订阅了 Topic A、Topic B、Topic C，则这三个 Topic 中的消息，被 Consumer Group 中的 Consumer 均匀消费。

Consumer Group 订阅多个 Topic 的示例代码如下：



```
String topicStr = kafkaProperties.getProperty("topic");
String[] topics = topicStr.split(",");
for (String topic: topics) {
subscribedTopics.add(topic.trim());
}
consumer.subscribe(subscribedTopics);
```

Topic 被多个 Consumer Group 订阅

一个 Topic 可以被多个 Consumer Group 订阅，且各个 Consumer Group 独立消费Topic下的所有消息。例如 Consumer Group A 订阅了 Topic A，Consumer Group B也订阅了Topic A，则发送到Topic A的每条消息，不仅会传

一份给Consumer Group A的消费实例，也会传一份给Consumer Group B的消费实例，且这两个过程相互独立，相互没有任何影响。

一个 Consumer Group 对应一个应用

建议一个 Consumer Group 对应一个应用，即不同的应用对应不同的代码。如果您需要将不同的代码写在同一个应用中，请准备多份不同的 kafka.properties。例如：kafka1.properties、kafka2.properties。

消费位点 Offset

每个 Topic 会有多个分区，每个分区会统计当前消息的总条数，这个称为最大位点 MaxOffset。
消息队列 CKafka 的 Consumer 会按顺序依次消费分区内的每条消息，记录已经消费了的消息条数，称为消费位点 ConsumerOffset。
剩余的未消费的条数（也称为消息堆积量）=MaxOffset-ConsumerOffset。

offset 提交

消息队列 CKafka 的 Consumer 有两个相关参数：
enable.auto.commit：默认值为true。
auto.commit.interval.ms： 默认值为5000，即5s。
这两个参数组合的结果为：每次 poll 数据前会先检查上次提交位点的时间，如果距离当前时间已经超过参数 auto.commit.interval.ms 规定的时长，则客户端会启动位点提交动作。
因此，如果将 enable.auto.commit 设置为 true，则需要在每次 poll 数据时，确保前一次 poll 出来的数据已经消费完毕，否则可能导致位点跳跃。
如果想自己控制位点提交，请把 enable.auto.commit 设为 false，并调用 commit(offsets) 函数自行控制位点提交。
注意：
尽量避免提交位点请求过于频繁，否则容易导致 Broker CPU 很高，影响正常的服务。例如自动提交位点设置 auto.commit.interval.ms 为100ms，手动提交位点，在高吞吐场景下，每消费一条消息提交一个位点。

重置 offset

以下两种情况，会发生消费位点重置：
当服务端不存在曾经提交过的位点时（例如客户端第一次上线）。
当从非法位点拉取消息时（例如某个分区最大位点是10，但客户端却从11开始拉取消息）。
Java 客户端可以通过 auto.offset.reset 来配置重置策略，主要有三种策略：
latest：从最大位点开始消费。
earliest：从最小位点开始消费。
none：不做任何操作，即不重置。
说明：
建议设置成 latest，而不要设置成 earliest，避免因位点非法时从头开始消费，从而造成大量重复。
如果是您自己管理位点，可以设置成 none。

拉取消息

消费过程是由客户端主动去服务端拉取消息的，在拉取大消息时需要控制拉取速度，注意以下参数设置：
max.poll.records：如果单条消息超过 1MB，建议设置为1。
max.partition.fetch.bytes：设置比单条消息的大小略大一点。
fetch.max.bytes：设置比单条消息的大小略大一点。
通过公网消费消息时，通常会因为公网带宽的限制导致连接被断开，此时需要注意控制拉取速度，注意以下参数设置：
fetch.max.bytes：建议设置成公网带宽的一半（注意该参数的单位是 bytes，公网带宽的单位是 bits）
max.partition.fetch.bytes：建议设置成 fetch.max.bytes 的三分之一或者四分之一。

拉取大消息

消费过程是由客户端主动去服务端拉取消息的，在拉取大消息时，需要注意控制拉取速度，注意修改配置：
max.poll.records：每次Poll获取的最大消息数量。如果单条消息超过1 MB，建议设置为1。
fetch.max.bytes：设置比单条消息的大小略大一点。
max.partition.fetch.bytes：设置比单条消息的大小略大一点。
拉取大消息的核心是逐条拉取的。

消息重复和消费幂等

消息队列 CKafka 消费的语义是 at least once， 也就是至少投递一次，保证消息不丢失，但是无法保证消息不重复。在出现网络问题、客户端重启时均有可能造成少量重复消息，此时应用消费端如果对消息重复比较敏感（例如订单交易类），则应该做消息幂等。
以数据库类应用为例，常用做法为：
发送消息时，传入 key 作为唯一流水号 ID。
消费消息时，判断 key 是否已经消费过，如果已经消费过了，则忽略，如果没消费过，则消费一次。
当然，如果应用本身对少量消息重复不敏感，则不需要做此类幂等检查。

消费失败

消息队列 CKafka 是按分区逐条消息顺序向前推进消费的，如果消费端拿到某条消息后执行消费逻辑失败，例如应用服务器出现了脏数据，导致某条消息处理失败，等待人工干预，那么有以下两种处理方式：
失败后一直尝试再次执行消费逻辑。这种方式有可能造成消费线程阻塞在当前消息，无法向前推进，造成消息堆积。
由于消息队列 CKafka 没有处理失败消息的设计，实践中通常会打印失败的消息或者存储到某个服务（例如创建一个 Topic 专门用来放失败的消息），然后定时检查失败消息的情况，分析失败原因，根据情况处理。

消费延迟

消费过程是由客户端主动去服务端拉取消息。一般情况下，如果客户端能够及时消费，则不会产生较大延迟。若产生了较大延迟，请先关注是否有堆积，并注意提高消费速度。

消费堆积

通常造成消息堆积的原因是：
消费速度跟不上生产速度，此时应该提高消费速度。
消费端产生了阻塞。
消费端拿到消息后，执行消费逻辑，通常会执行一些远程调用，如果这个时候同步等待结果，则有可能造成一直等待，消费进程无法向前推进。
消费端应该尽量避免堵塞消费线程，如果存在等待调用结果的情况，建议设置等待的超时时间，超时后作为消费失败进行处理。

提高消费速度

增加 Consumer 实例个数提高并行处理能力，如果消费者和分区数已经1:1，可以考虑增加分区数（注意：对于 flink 自动维护分区的场景不会自动感知新增分区后可能需要修改相关代码后重启）。 可以在进程内直接增加（需要保证每个实例对应一个线程），也可以部署多个消费实例进程。
说明：
实例个数超过分区数量后就不再能提高速度，将会有消费实例不工作。
增加消费线程。
1. 定义一个线程池。
2. Poll 数据。
3. 把数据提交到线程池进行并发处理。
4. 等并发结果返回成功后，再次 poll 数据执行。

套接字缓冲区（socket buffers）

在 Kafka 的 0.10.x 版本中，参数 receive.buffer.bytes 的默认值为 64KB。而在 Kafka 的 0.8.x 版本中，参数 socket.receive.buffer.bytes 的默认值为 100KB。
这两个默认值对于高吞吐量的环境而言都太小了，特别是如果 Broker 和 Consumer 之间的网络带宽延迟积（bandwidth-delay product）大于局域网（local areanetwork，LAN）时。
对于延迟为1ms或更多的高带宽的网络（例如 10Gbps 或更高），建议将套接字缓冲区设置为8或16MB。
如果您的内存不足，也至少考虑设置为 1MB。您也可以设置为 -1，它会让底层操作系统根据网络的实际情况，去调整缓冲区的大小。
但是，对于需要启动“热”分区的 Consumers 来说，自动调整可能不会那么快。

消息广播

CKafka 目前没有消息广播的语义，可以通过创建不同的 Group 来模拟实现。

消息过滤

CKafka 自身没有消息过滤的语义。实践中可以采取以下两个办法：
如果过滤的种类不多，可以采取多个 Topic 的方式达到过滤的目的。
如果过滤的种类多，则最好在客户端业务层面自行过滤。
实践中请根据业务具体情况进行选择，也可以综合运用上面两种办法。

消费某些分区不消费

消费者在消费过程中，可能遇到消费者在线，但是某些分区的位点一致不前进，可能原因如下：

1. 遇到一条异常消息，可能是超大消息，格式异常，导致消费者拉取消息时候，转换成业务位点。
2. 使用公网带宽，带宽较小，拉取大消息时候直接把带宽打满，导致在超时时间内拉取不到消息。
3. 消费者假死，导致不去拉取。

解决方式：

关掉消费者，在 CKafka 控制台设置位点，跳过某些异常消息，或者优化消费代码，然后重启消费者消费。

Confluent Go SDK

最近更新时间：2024-07-04 16:01:00

背景

TDMQ CKafka 是一个分布式流处理平台，用于构建实时数据管道和流式应用程序。它具备高吞吐量、低延迟、可伸缩性和容错性等特性。

Sarama：Shopify 开发的一个 Kafka 库，提供了生产者、消费者、分区消费者等功能。该库的性能较好，社区支持也较为活跃。

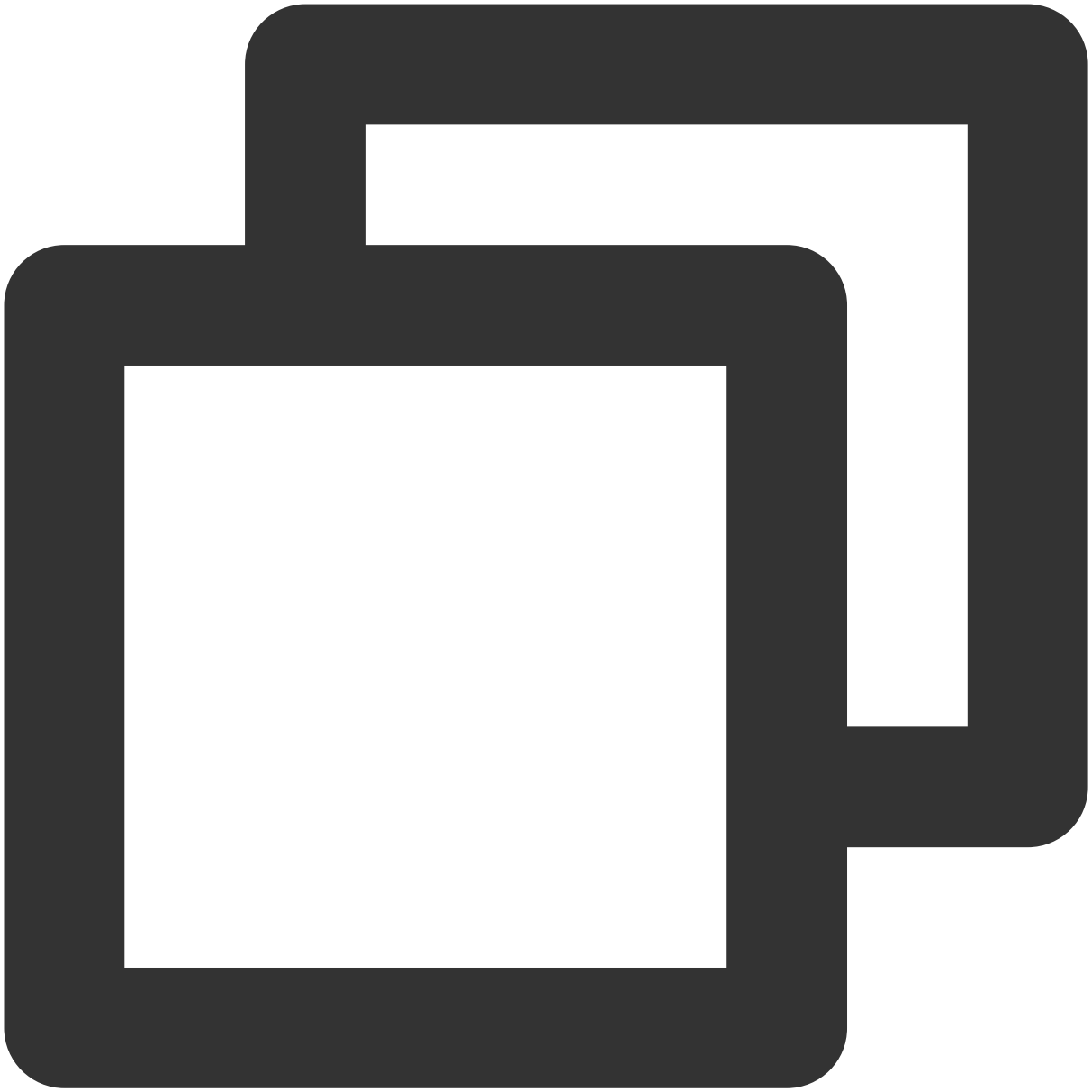
Confluent-Kafka-Go：由 Confluent 开发的 Kafka 库，提供了高级 API，易于使用。该库基于 librdkafka C 库，性能非常优秀，但安装和使用略显复杂。

本文着重介绍上述 Confluent Go 客户端的关键参数、实践教程以及常见问题。

生产者实践

版本选择

在使用 Confluent Go SDK 时，可以通过配置参数 "bootstrap.servers" 来指定 Kafka 集群的地址，而 Broker 的版本则可以通过"api.version.request"参数来设置，这样 Confluent Go SDK 会在启动时自动检测 Broker 的版本。

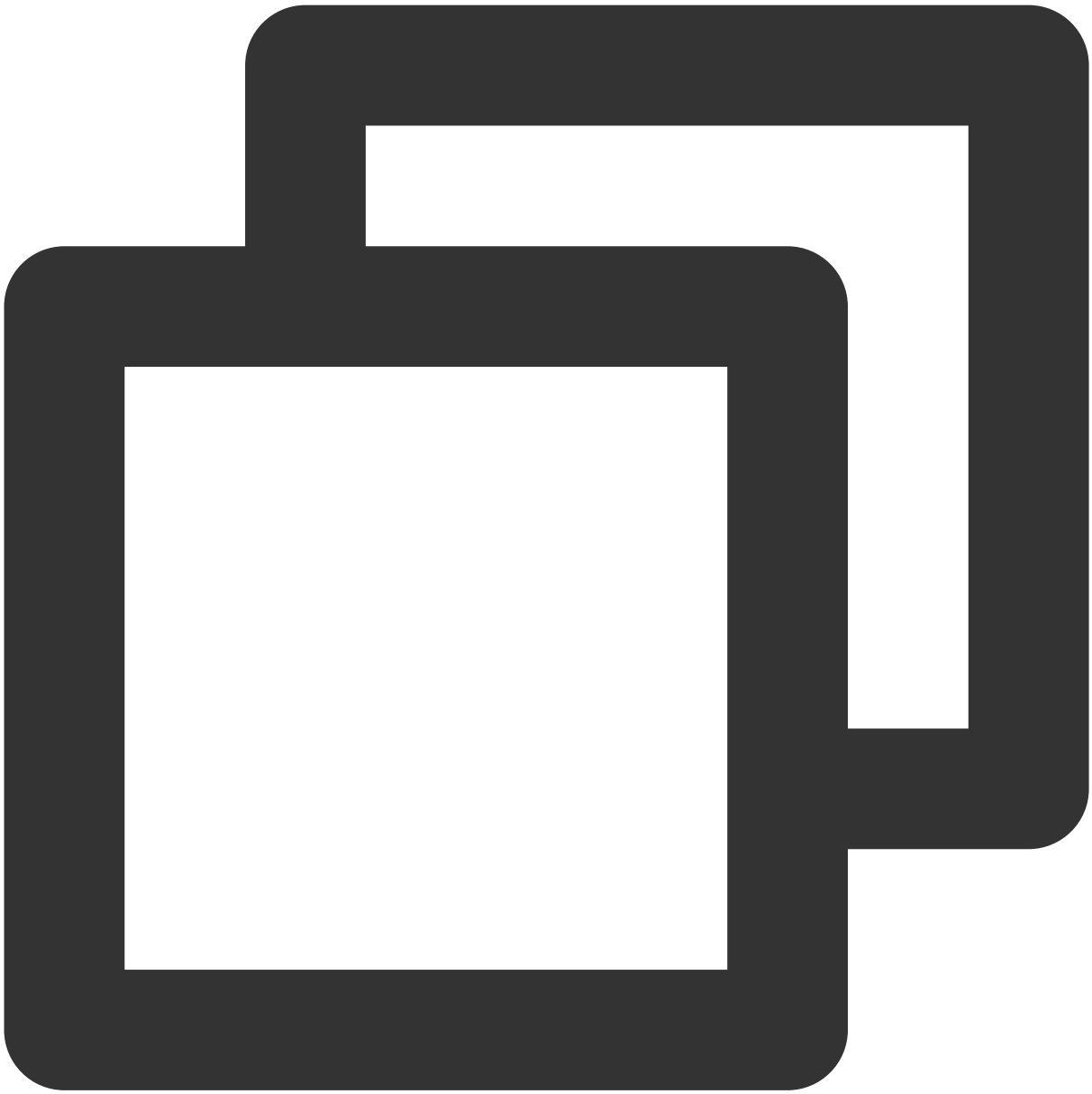


```
config := &kafka.ConfigMap{
    "bootstrap.servers": "localhost",
    "api.version.request": true,
}
```

生产者参数与调优

生产者参数

Confluent Go 是基于 librdkafka 开发的，在使用 Confluent Go 客户端写入 Kafka 的时候，需要配置的参数会透传 librdkafka，主要涉及如下关键参数，相关的参数和默认值如下：



```
package main

import (
    "fmt"
    "github.com/confluentinc/confluent-kafka-go/kafka"
)
```

```
func main() {
    config := &kafka.ConfigMap{
                "bootstrap.servers":                "localhost:9092",
        "acks":                             -1,          //ack方式，默认值为-1
                "client.id":                        "rdkafka",    //客户端ID
                "compression.type":                 "none",      //指定压缩方式
        "compression.level":                -1,          //压缩等级
                "batch.num.messages":               10000,       //默认一个批次最多聚合
        "batch.size":                       1000000,     //构成MessageSet整批大小限制，
        "queue.buffering.max.ms":           5,           //在构造消息批次(MessageSets)
                "queue.buffering.max.messages":     100000,      //Producer攒批发送中，
                "queue.buffering.max.kbytes":       1048576,     //Producer攒批发送中，
                "message.send.max.retries":         2147483647, //重试次数，默认214748
                "retry.backoff.ms":                 100,        //重试间隔时间，默认100
                "socket.timeout.ms":                60000,      //会话超时时间，默认60s

    }

    producer, err := kafka.NewProducer(config)
    if err != nil {
                panic(fmt.Sprintf("Failed to create producer: %s", err))
    }

    // 使用producer发送消息等操作...

    // 关闭producer
    producer.Close()
}
```

参数说明调优

关于 acks 参数优化

acks 参数用于控制生产者发送消息时的确认机制。该参数的默认值为-1，表示消息发送给 Leader Broker 后，Leader 确认以及相应的 Follower 消息都写入完成后才返回。acks 参数还有以下可选值：0，1，-1。在跨可用区场景，以及副本数较多的 Topic，acks 参数的取值会影响消息的可靠性和吞吐量。

在一些在线业务消息的场景下，吞吐量要求不大，可以将 acks 参数设置为-1，确保消息被所有副本接收和确认后才返回，从而提高消息的可靠性。

在日志采集等大数据或者离线计算的场景下，要求高吞吐（即每秒写入 Kafka 的数据量）的情况下，可以将 acks 设置为1，提高吞吐。

关于 buffering 参数优化（缓存）

默认情况下，传输同等数据量的情况下，多次请求和一次请求的网络传输，一次请求传输能有效减少相关计算和网络资源，提高整体写入的吞吐量。

因此，可以通过这个参数设置优化客户端发送消息的吞吐能力。对于 Confluent kafka Go，默认提供5ms的攒批时间积攒消息。如果消息较小，可以适当增加queue.buffering.max.ms的时间。

关于压缩参数优化

Confluent Go 支持如下压缩参数：none, gzip, snappy, lz4, zstd。

在 Confluent Kafka Go 客户端中，支持以下几种压缩算法：

none：不使用压缩算法。

gzip：使用 GZIP 压缩算法。

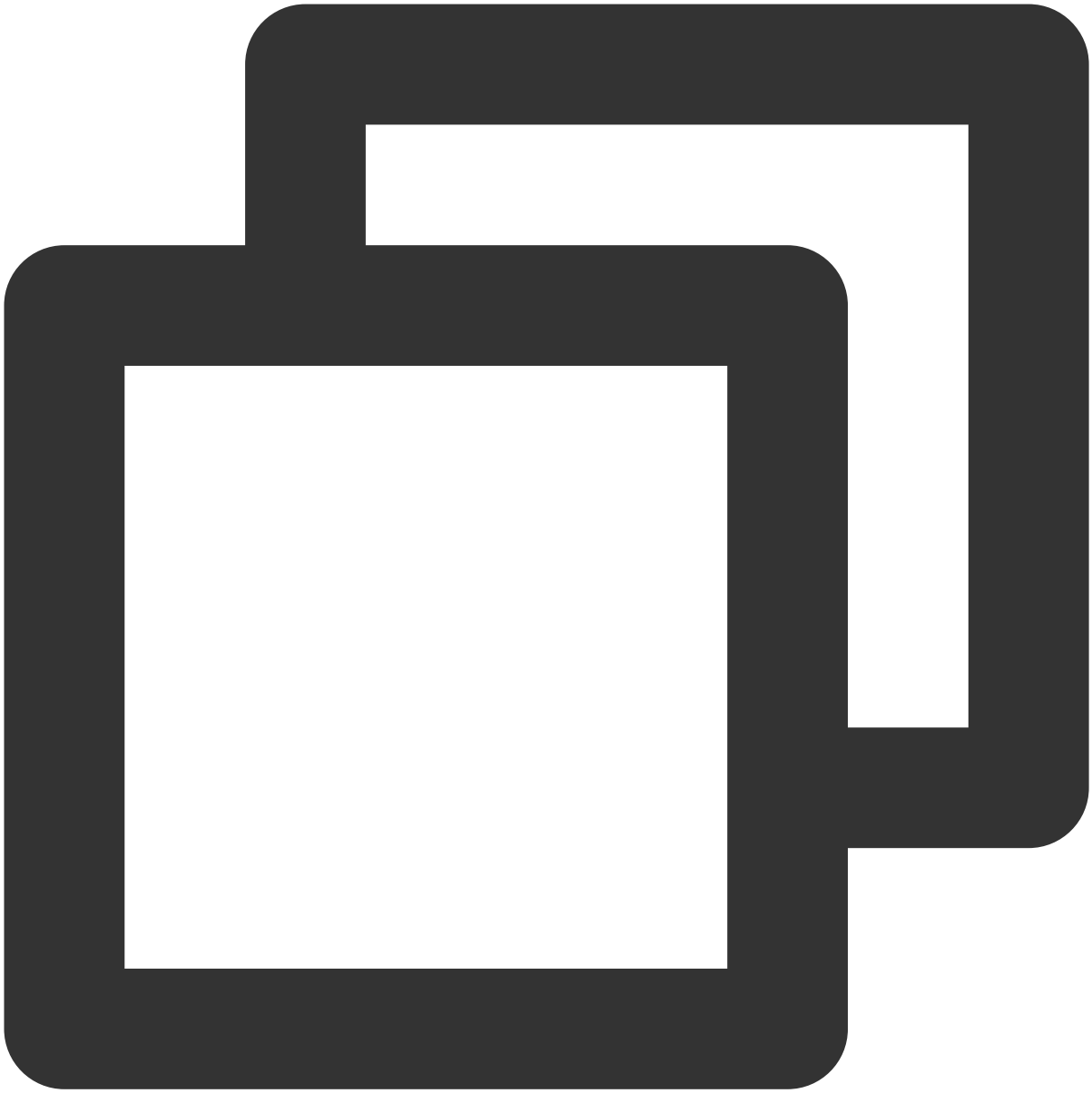
snappy：使用 Snappy 压缩算法。

lz4：使用 LZ4 压缩算法。

zstd：使用 ZSTD 压缩算法。

要在 Producer 客户端中使用压缩算法，需要在创建生产者时设置 compression.type 参数。例如，要使用LZ4压缩算法，可以将 compression.type 设置为 lz4，虽然压缩算法的 CPU 压缩，和 CPU 解压缩，发生客户端，是一种用计算换带宽的优化方式，但是由于 Broker 针对压缩消息存在校验行为会付出额外的计算成本，尤其是 Gzip 压缩，服务端的压缩计算成本会比较大，在某种程度上可能会出现得不偿失的情况，反而因为计算的增加导致 Broker 消息处理能力偏低，导致带宽吞吐更低。这种情况建议可以使用如下方式进行使用：

1. 在 Producer 端对消息数据独立压缩，生成压缩包数据：messageCompression，同时在消息的 key 存储压缩方式：



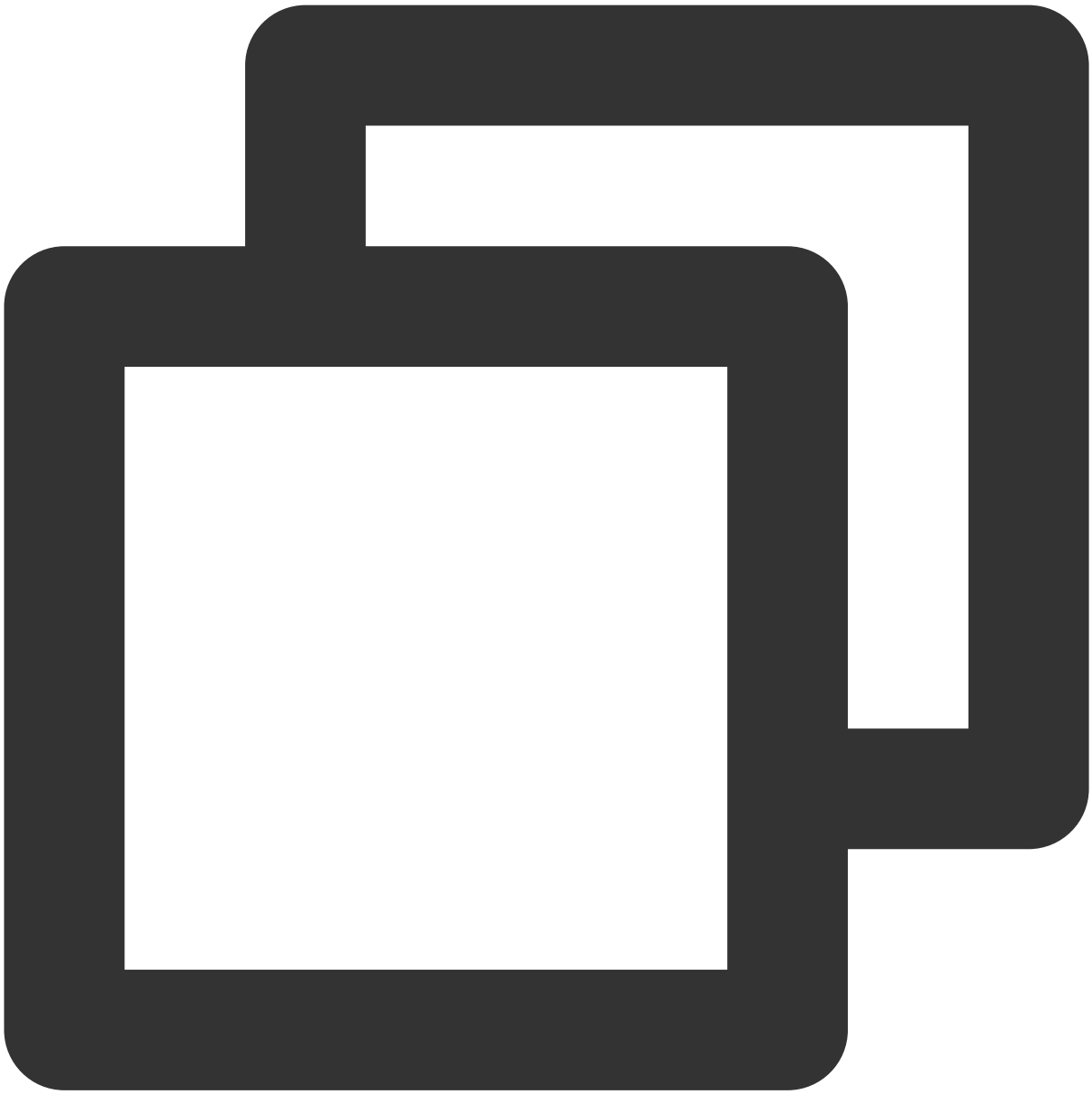
```
{"Compression","CompressionLZ4"}
```

2. 在 Producer 端将 messageCompression 当成正常消息发送。
3. 在 Consumer 端读取消息 key，获取使用的压缩方式，独立进行解压缩。

创建生产者实例

如果应用程序需要更高的吞吐量，则可以使用异步生产者，以提高消息的发送速度。同时，可以使用批量发送消息的方式，以减少网络开销和 IO 消耗。如果应用程序需要更高的可靠性，则可以使用同步生产者，以确保消息发送成

功。同时，可以使用 ACK 确认机制和事务机制，以确保消息的可靠性和一致性。具体的参数调优参考生产者参数与调优。



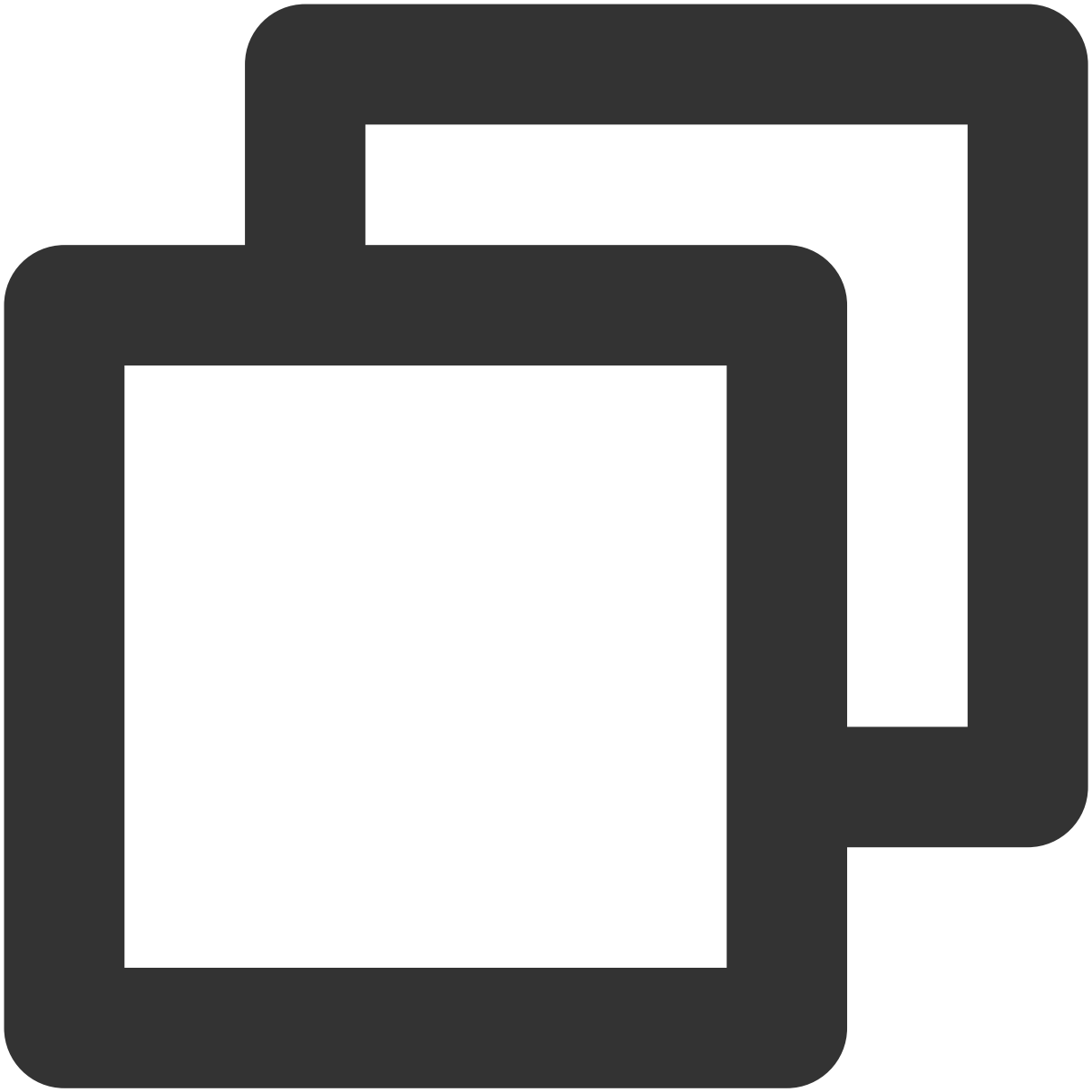
```
package main

import (
    "fmt"
    "github.com/confluentinc/confluent-kafka-go/kafka"
)
```

```
func main() {
    // 配置Kafka Producer
    p, err := kafka.NewProducer(&kafka.ConfigMap{
        "bootstrap.servers": "localhost:9092",
        "acks":              "1",
        "compression.type":  "none",
        "batch.num.messages": "1000",
    })
    if err != nil {
        fmt.Printf("Failed to create producer: %s\\n", err)
        return
    }

    // 发送消息
    for i := 0; i < 10; i++ {
        topic := "test-topic"
        value := fmt.Sprintf("hello world %d", i)
        message := &kafka.Message{
            TopicPartition: kafka.TopicPartition{Topic: &topic, Partition: kafka.Pa
            Value:          []byte(value),
        }
        p.Produce(message, nil)
    }

    // 关闭Kafka Producer
    p.Flush(15 * 1000)
    p.Close()
}
```

消费者实践

消费者参数与调优

消费者参数



```
package main

import (
    "fmt"
    "github.com/confluentinc/confluent-kafka-go/kafka"
)

func main() {
    // 配置Kafka Consumer
    c, err := kafka.NewConsumer(&kafka.ConfigMap{
```

```
        "bootstrap.servers": "localhost:9092",
        "group.id":          "test-group",
        "auto.offset.reset":"earliest",
        "fetch.min.bytes":1, //最小拉取字节数
        "fetch.max.bytes":52428800,//最大拉取字节数
        "fetch.wait.max.ms":"500",  //如果没有最新消费消息默认等待500ms
        "enable.auto.commit":true, //是否支持自动提交位点，默认true
        "auto.commit.interval.ms":5000,//自动提交位点间隔，默认5s
        "max.poll.interval.ms": 300000,//Consumer 在两次 poll 操作之间的最大延迟。默认5分
        "session.timeout.ms": 45000,//session时间，默认45s
        "heartbeat.interval.ms": 3000,//心跳时间，默认3s
    })
    if err != nil {
        fmt.Printf("Failed to create consumer: %s\\n", err)
        return
    }

    // 订阅主题
    c.SubscribeTopics([]string{"test-topic"}, nil)

    // 手动提交位点
    for {
        ev := c.Poll(100)
        if ev == nil {
            continue
        }

        switch e := ev.(type) {
        case *kafka.Message:
            fmt.Printf("Received message: %s\\n", string(e.Value))
            c.CommitMessage(e)
        case kafka.Error:
            fmt.Printf("Error: %v\\n", e)
        }
    }

    // 关闭Kafka Consumer
    c.Close()
}
```

参数说明与调优

1. max.poll.interval.ms 是 Kafka Consumer 的一个配置参数，它用于指定 Consumer 在两次 poll 操作之间的最大延迟。这个参数的主要作用是控制 Consumer 的 liveness，也就是判断 Consumer 是否还活着。如果 Consumer 在 max.poll.interval.ms 指定的时间内没有进行 poll 操作，那么 Kafka 认为这个 Consumer 已经挂掉，会触发 Consumer 的 rebalance 操作。这个参数的设置需要根据实际的消费速度来调整。如果设置得太小，可能会导致 Consumer 频繁

地触发 rebalance 操作，增加了 Kafka 的负担；如果设置得太大，可能会导致 Consumer 在出现问题时不能及时被 Kafka 检测到，从而影响了消息的消费。

2. 一般消费主要是 rebalance 时间频繁和消费线程阻塞问题，参考以下说明参数优化：

2.1 session.timeout.ms：v0.10.2之前的版本可适当提高该参数值，需要大于消费一批数据的时间，但不要超过30s，建议设置为25s；而v0.10.2及其之后的版本，保持默认值10s即可。

2.2 heartbeat.interval.ms：默认3s，设置该值 需要小于session.timeout.ms/3。

2.3 max.poll.interval.m：默认5分钟，如果分区数和消费者较多，建议适当调大该值。该值要大于<max.poll.records> / (<单个线程每秒消费的条数> * <消费线程的个数>)的值。

注意：

如果消息处理是同步处理，即拉取消息、处理、再拉取下一个消息，需要做如下改造：

根据需求调大 MaxProcessingTime 时间。

针对处理时间大于 MaxProcessingTime 请求处理时间进行监控，采样打印超时时间。

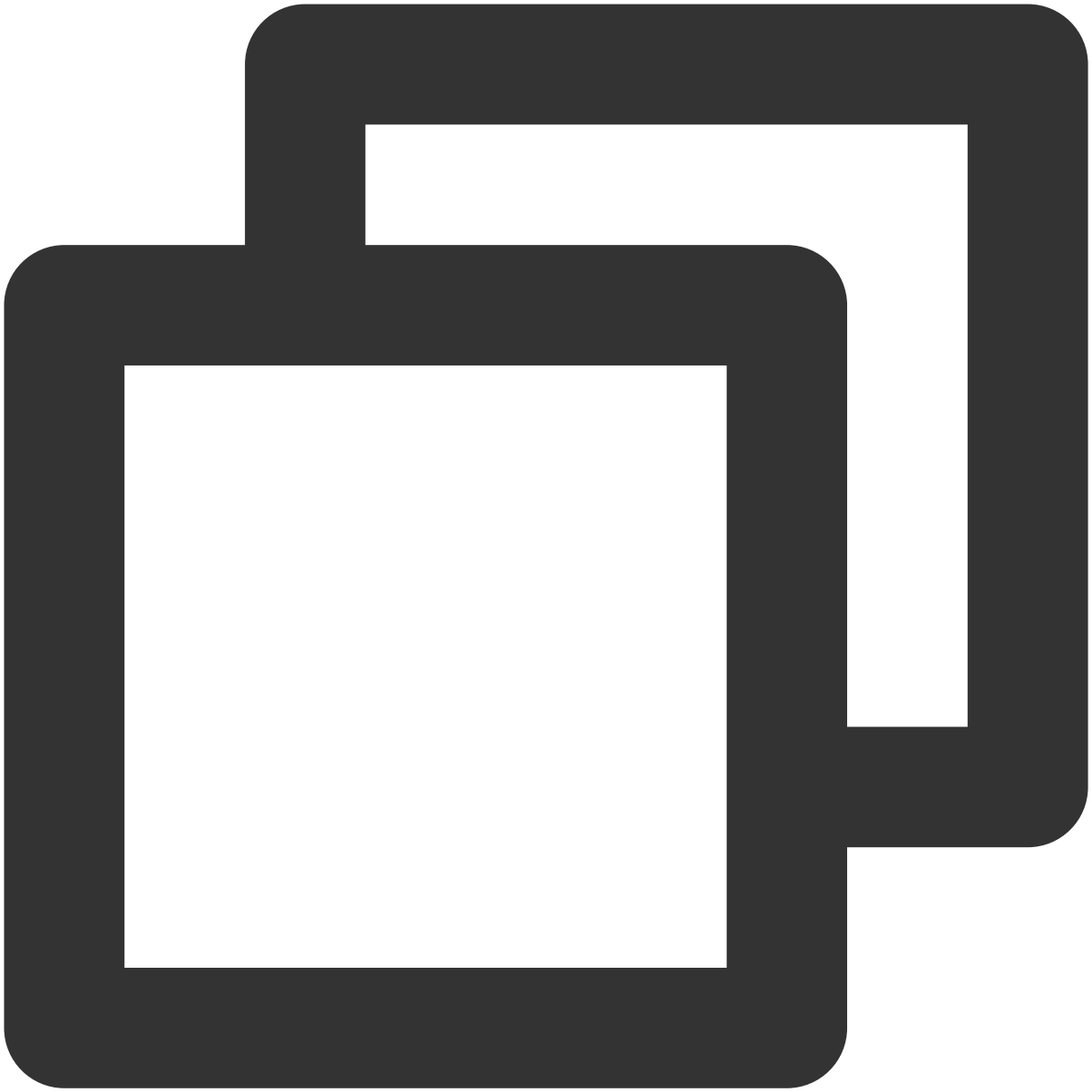
3. 针对自动提交位点请求，建议 auto.commit.interval.ms 时间不要低于1000ms，因为频率过高的位点请求会导致 Broker CPU 很高，影响其他正常服务的读写。

创建消费者实例

Confluent Go 提供订阅的模型创建消费者，其中在提交位点方面，提供手动提交位点和自动提交位点两种方式。

自动提交位点

自动提交位点：消费者在拉取消息后会自动提交位点，无需手动操作。这种方式的优点是简单易用，但是可能会导致消息重复消费或丢失。

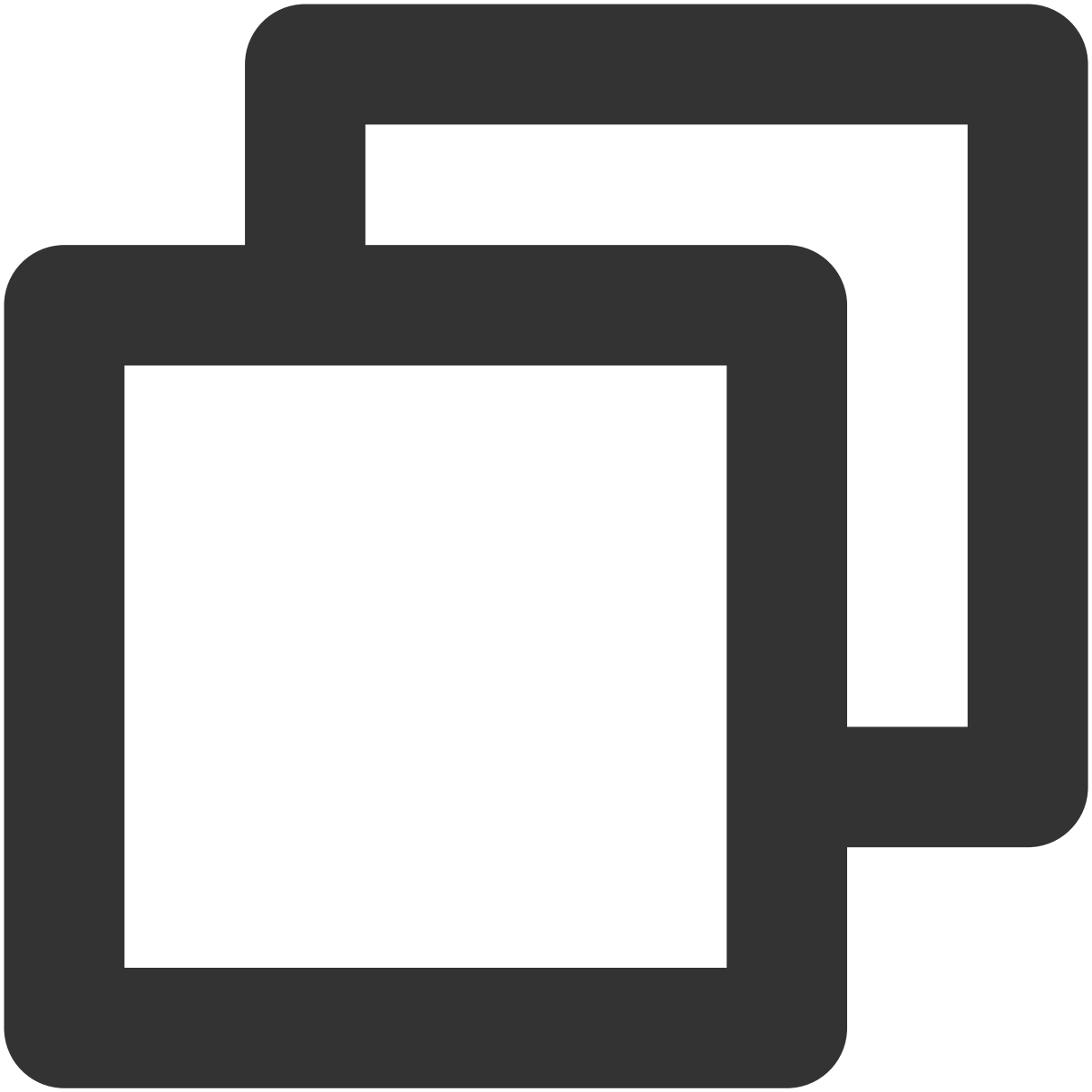


```
package main

import (
    "fmt"
    "github.com/confluentinc/confluent-kafka-go/kafka"
)

func main() {
    // 配置Kafka Consumer
    c, err := kafka.NewConsumer(&kafka.ConfigMap{
```

```
        "bootstrap.servers": "localhost:9092",
        "group.id":          "test-group",
        "auto.offset.reset": "earliest",
        "enable.auto.commit": true,     //是否启用自动提交位点。设置为true，表示启用自动提交
        "auto.commit.interval.ms": 5000, //自动提交位点的间隔时间。设置为5000毫秒（即5秒）
        "max.poll.interval.ms": 300000,//Consumer在一次poll操作中最长的等待时间。设置为30
        "session.timeout.ms": 10000,//指定Consumer与broker之间的会话超时时间,设置10秒
        "heartbeat.interval.ms": 3000, //指定Consumer发送心跳消息的间隔时间。设置为3000毫
    })
    if err != nil {
        fmt.Printf("Failed to create consumer: %s\\n", err)
        return
    }

    // 订阅主题
    c.SubscribeTopics([]string{"test-topic"}, nil)

    // 自动提交位点
    for {
        ev := c.Poll(100)
        if ev == nil {
            continue
        }

        switch e := ev.(type) {
        case *kafka.Message:
            fmt.Printf("Received message: %s\\n", string(e.Value))
        case kafka.Error:
            fmt.Printf("Error: %v\\n", e)
        }
    }

    // 关闭Kafka Consumer
    c.Close()
```

手动提交位点

手动提交位点：消费者在处理完消息后需要手动提交位点。这种方式的优点是可以精确控制位点的提交，避免消息重复消费或丢失。但是需要注意，手动提交位点如果太频繁会导致 Broker CPU 很高，影响性能，随着消息量增加，CPU 消费会很高，影响正常 Broker 的其他功能，因此建议间隔一定消息提交位点。



```
package main

import (
    "fmt"
    "github.com/confluentinc/confluent-kafka-go/kafka"
)

func main() {
    // 配置Kafka Consumer
    c, err := kafka.NewConsumer(&kafka.ConfigMap{
```

```
        "bootstrap.servers": "localhost:9092",
        "group.id":          "test-group",
        "auto.offset.reset": "earliest",
        "enable.auto.commit": false,
        "max.poll.interval.ms": 300000,
        "session.timeout.ms": 10000,
        "heartbeat.interval.ms": 3000,
    })
    if err != nil {
        fmt.Printf("Failed to create consumer: %s\\n", err)
        return
    }

    // 订阅主题
    c.SubscribeTopics([]string{"test-topic"}, nil)

    // 手动提交位点
    for {
        ev := c.Poll(100)
        if ev == nil {
            continue
        }

        switch e := ev.(type) {
        case *kafka.Message:
            fmt.Printf("Received message: %s\\n", string(e.Value))
            c.CommitMessage(e)
        case kafka.Error:
            fmt.Printf("Error: %v\\n", e)
        }
    }

    // 关闭Kafka Consumer
    c.Close()
```

Sarama Go

最近更新时间：2024-07-04 16:00:59

背景

TDMQ CKafka 是一个分布式流处理平台，用于构建实时数据管道和流式应用程序。它具备高吞吐量、低延迟、可伸缩性和容错性等特性。
Sarama：Shopify 开发的一个 Kafka 库，提供了生产者、消费者、分区消费者等功能。该库的性能较好，社区支持也较为活跃。
Confluent-Kafka-Go：由 Confluent 开发的 Kafka 库，提供了高级 API，易于使用。该库基于 librdkafka C 库，性能非常优秀，但安装和使用略显复杂。
本文着重介绍上述 Sarama Go 客户端的关键参数、实践教程以及常见问题。

生产者实践

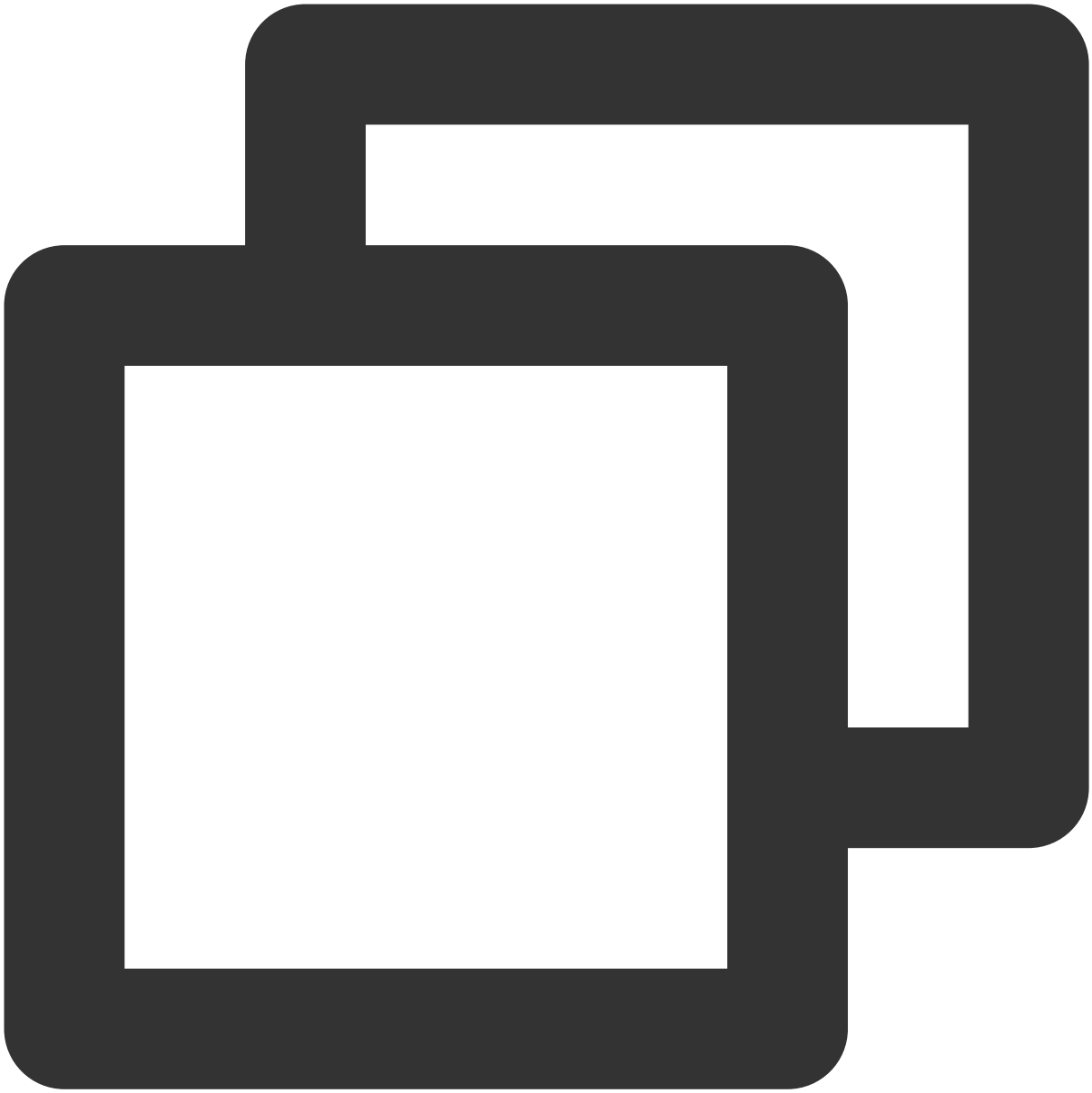
版本选择

在选择 Sarama 客户端版本时，需要确保所选版本与 Kafka broker 版本兼容。Sarama 库支持多个 Kafka 协议版本，可以通过设置 config.Version 来指定使用的协议版本。常见的 Kafka 协议版本与 Sarama 库版本的对应关系如下，目前最新版本请参见 Sarama 版本。

Kafka 版本	Sarama 库版本	Sarama 协议版本常量
0.8.2.x	>= 1.0.0	sarama.V0_8_2_0
0.9.0.x	>= 1.0.0	sarama.V0_9_0_0
0.10.0.x	>= 1.0.0	sarama.V0_10_0_0
0.10.1.x	>= 1.0.0	sarama.V0_10_1_0
0.10.2.x	>= 1.0.0	sarama.V0_10_2_0
0.11.0.x	>= 1.16.0	sarama.V0_11_0_0
1.0.x	>= 1.16.0	sarama.V1_0_0_0
1.1.x	>= 1.19.0	sarama.V1_1_0_0
2.0.x	>= 1.19.0	sarama.V2_0_0_0
2.1.x	>= 1.21.0	sarama.V2_1_0_0

2.2.x	>= 1.23.0	sarama.V2_2_0_0
2.3.x	>= 1.24.0	sarama.V2_3_0_0
2.4.x	>= 1.27.0	sarama.V2_4_0_0
2.5.x	>= 1.28.0	sarama.V2_5_0_0
2.6.x	>= 1.29.0	sarama.V2_6_0_0
2.7.x	>= 1.29.0	sarama.V2_7_0_0
2.8.x以及以上	建议使用>=1.42.1	sarama.V2_8_0_0-sarama.V3_6_0_0

上述列出的 Sarama 库版本是支持对应 Kafka 协议版本的最低版本。为了获得最佳性能和使用新功能，建议使用 Sarama 的最新版本。在使用最新版本时，客户可以通过设置 config.Version 来指定与您的 Kafka broker 兼容的协议版本。设置方式如下，务必先设置版本后使用，否则会有预期外的不兼容问题：

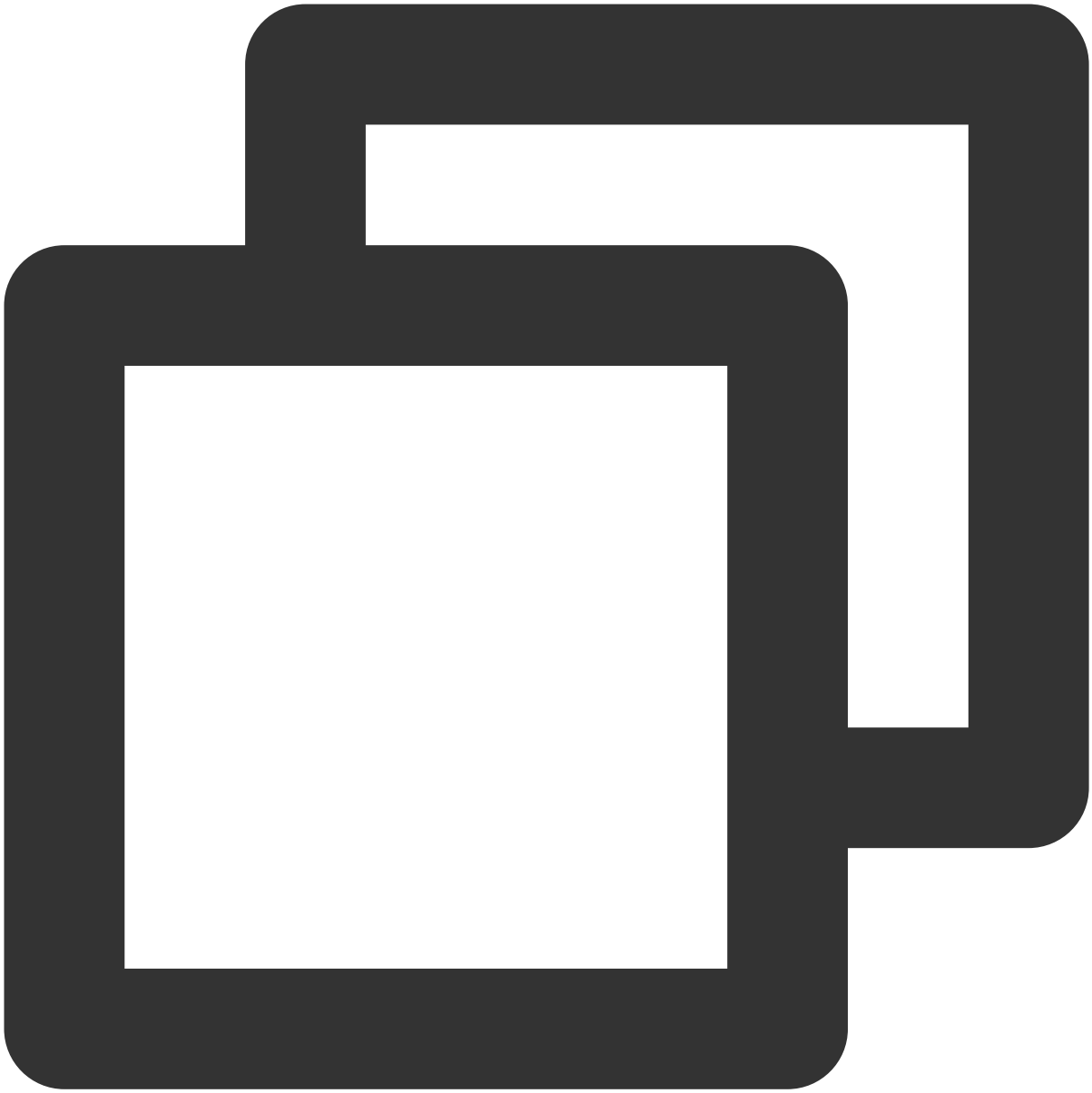


```
config := sarama.NewConfig()
config.Version = sarama.V2_7_0_0 // 根据实际Kafka版本设置协议版本
```

生产者参数与调优

生产者参数

在使用 Sarama go 客户端写入 kafka 时候，需要配置如下关键参数，相关的参数和默认值如下：



```
config := sarama.NewConfig()
sarama.MaxRequestSize = 100 * 1024 * 1024  //请求最大大小，默认100MB，可以调整，写入大于1
sarama.MaxResponseSize = 100 * 1024 * 1024 //响应最大大小，默认100MB，可以调整，获取大于1

config.Producer.RequiredAcks = sarama.WaitForLocal // 默认值为sarama.WaitForLocal(1)

config.Producer.Retry.Max = 3                      // 生产者重试的最大次数，默认为3
config.Producer.Retry.Backoff = 100 * time.Millisecond // 生产者重试之间的等待时间，默认

config.Producer.Return.Successes = false          //是否返回成功的消息，默认为false
config.Producer.Return.Errors = true             // 返回失败的消息，默认值为true
```

```
config.Producer.Compression = CompressionNone  //对消息是否压缩后发送，默认CompressionN
config.Producer.CompressionLevel = CompressionLevelDefault // 指定压缩等级，在配置了压缩

config.Producer.Flush.Frequency = 0 //producer缓存消息的时间，默认缓存0毫秒
config.Producer.Flush.Bytes = 0       // 达到多少字节时，触发一次broker请求，默认为0，直接
config.Producer.Flush.Messages = 0      // 达到多少条消息时，强制，触发一次broker请求，这
config.Producer.Flush.MaxMessages = 0       // 最大缓存多少消息，默认为0，有消息立刻发送，M

config.Producer.Timeout = 5 * time.Second       // 超时时间

config.Producer.Idempotent = false                    //是否需要幂等，默认false
config.Producer.Transaction.Timeout = 1 * time.Minute // 事务超时时间默认1分钟
config.Producer.Transaction.Retry.Max = 50            //事务重试时间
config.Producer.Transaction.Retry.Backoff = 100 * time.Millisecond
config.Net.MaxOpenRequests = 5                        //默认值5，一次发送请求的数量
config.Producer.Transaction.ID = "test"               //事务ID

config.ClientID = "your-client-id"              // 客户端ID
```

参数说明调优

关于 RequiredAcks 参数优化

RequiredAcks 参数用于控制生产者发送消息时的确认机制。该参数的默认值为 WaitForLocal，表示消息发送给 Leader Broker 后，Leader 确认消息写入后即返回。RequiredAcks 参数还有以下可选值：

NoResponse: 不等待任何确认，直接返回。

WaitForLocal: 等待 Leader 副本确认写入后返回。

WaitForAll: 等待 Leader 副本以及相关的 Follower 副本确认写入后返回。

由上可知，在跨可用区场景，以及副本数较多的 Topic，RequiredAcks 参数的取值会影响消息的可靠性和吞吐量。因此：

在一些在线业务消息的场景下，吞吐量要求不大，可以将 RequiredAcks 参数设置为 WaitForAll，则可以确保消息被所有副本接收和确认后才返回，从而提高消息的可靠性。

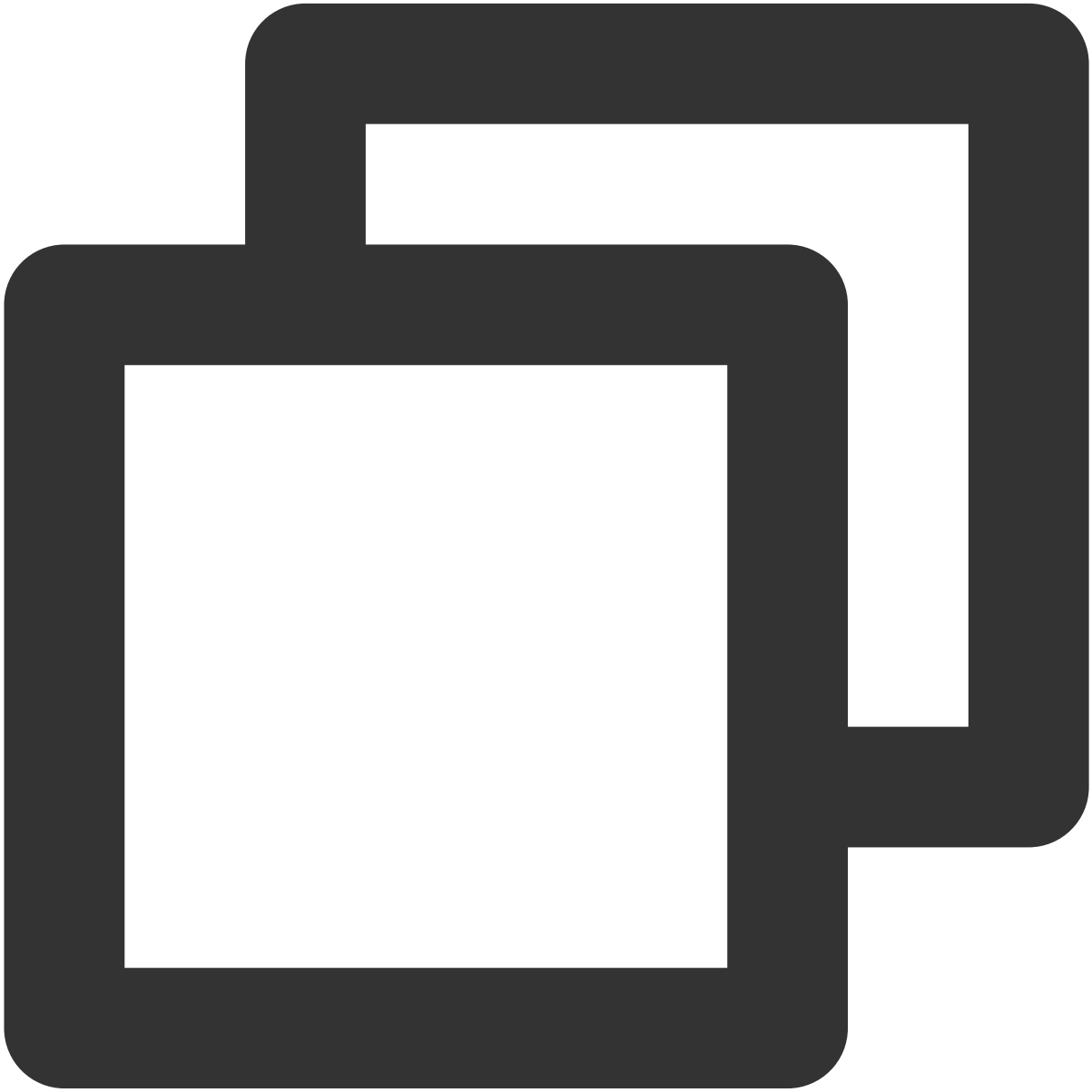
在日志采集等大数据或者离线计算的场景下，要求高吞吐（即每秒写入 Kafka 的数据量）的情况下，可以将 RequiredAcks 设置为 WaitForLocal，提高吞吐。

关于 Flush 参数优化（缓存）

默认情况下，传输同等数据量的情况下，多次请求和一次请求的网络传输，一次请求传输能有效减少相关计算和网络资源，提高整体写入的吞吐量。因此，可以通过这个参数设置优化客户端发送消息的吞吐能力。在高吞吐场景下，可以配合计算和设置：

其中 Bytes 建议设置为16K，对齐 Kafka 标准 Java SDK 定义， 预估一条消息为1K（1024）个字节，因此得出如下 Messages 和 MaxMessages 的写入参数：

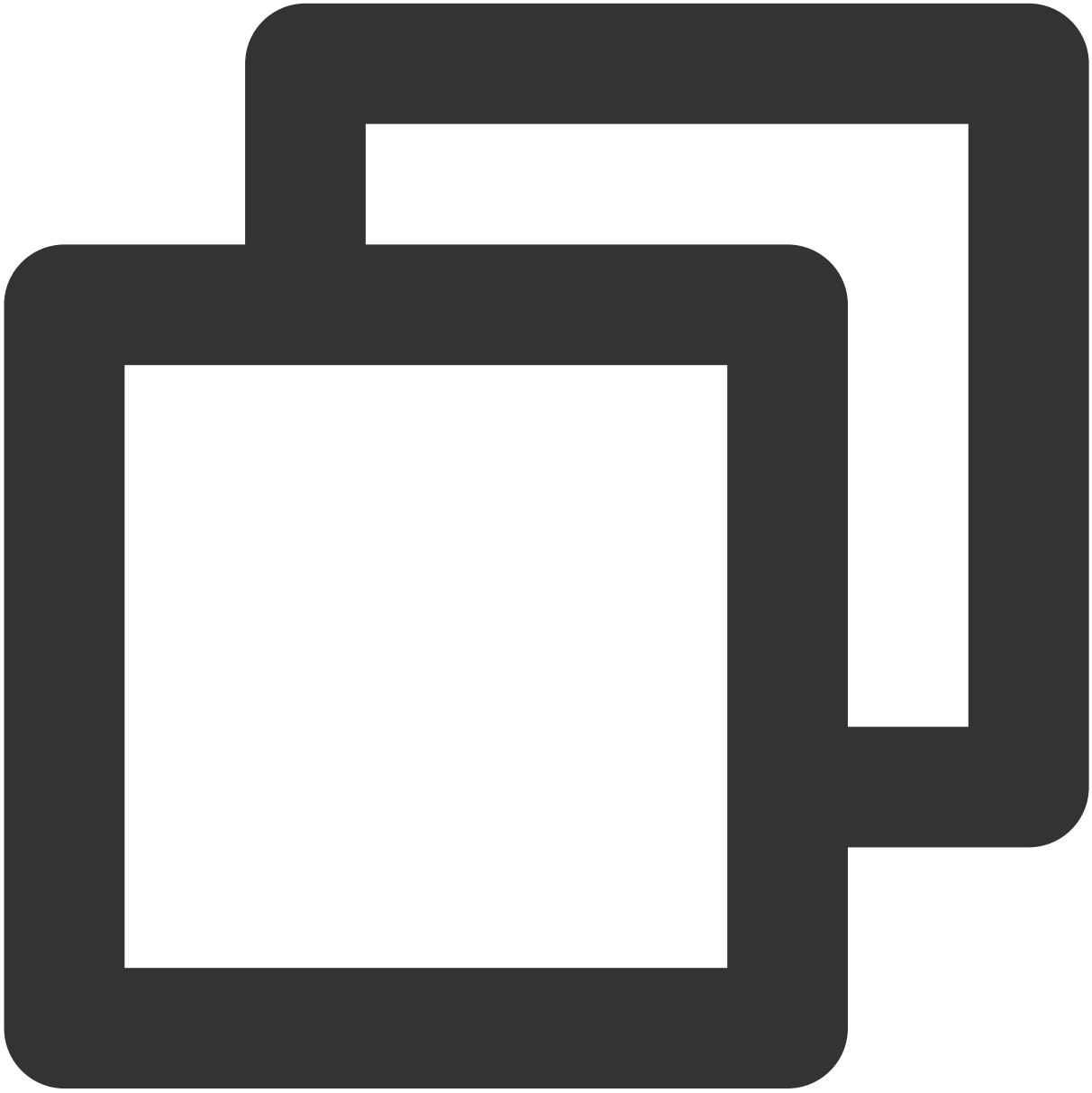
其中 Frequency 的计算方式为：预估流量为 16MB，分区数为16个分区，此时单分区每秒写入流量为：16 * 1024 * 1024 / 16 = 1 * 1024 * 1024 = 1MB，单个分区每秒 1MB 的流量。假设按照 16K 一个请求,数据量发送，那么在 1s 内要实现 1MB 的流量传输 1*1024*1024/16/1024 = 64个请求，因此 Frequency <= 15.62ms(1000/64)。
实际上，由于业务流量不是持续生产的，在低峰期，可能出现即时达到 16ms，也缓存不了太多的数据，因此在高吞吐的情况下，可以将条件简化，以 Bytes 为准，Frequency 可以适当调大，例如能接受 500ms 的延时增加，那么就可以设置为 500ms，因为此时如果命中数据量大于等于 Bytes，会按照 Bytes 的条件发送请求。



```
config.Producer.Flush.Frequency = 16 //producer缓存消息的时间， 默认缓存100毫秒,如果发送的
config.Producer.Flush.Bytes = 16*1024        // 达到多少字节时，触发一次broker请求，默认为
```

```
config.Producer.Flush.Messages = 17       // 达到多少条消息时，强制，触发一次broker请求，这
config.Producer.Flush.MaxMessages = 16   // 16条，实际上因为消息大小不严格1024字节，Messag
                                          //因为命中Frequency，Bytes，MaxMessages < Mes
```

关于事务参数优化



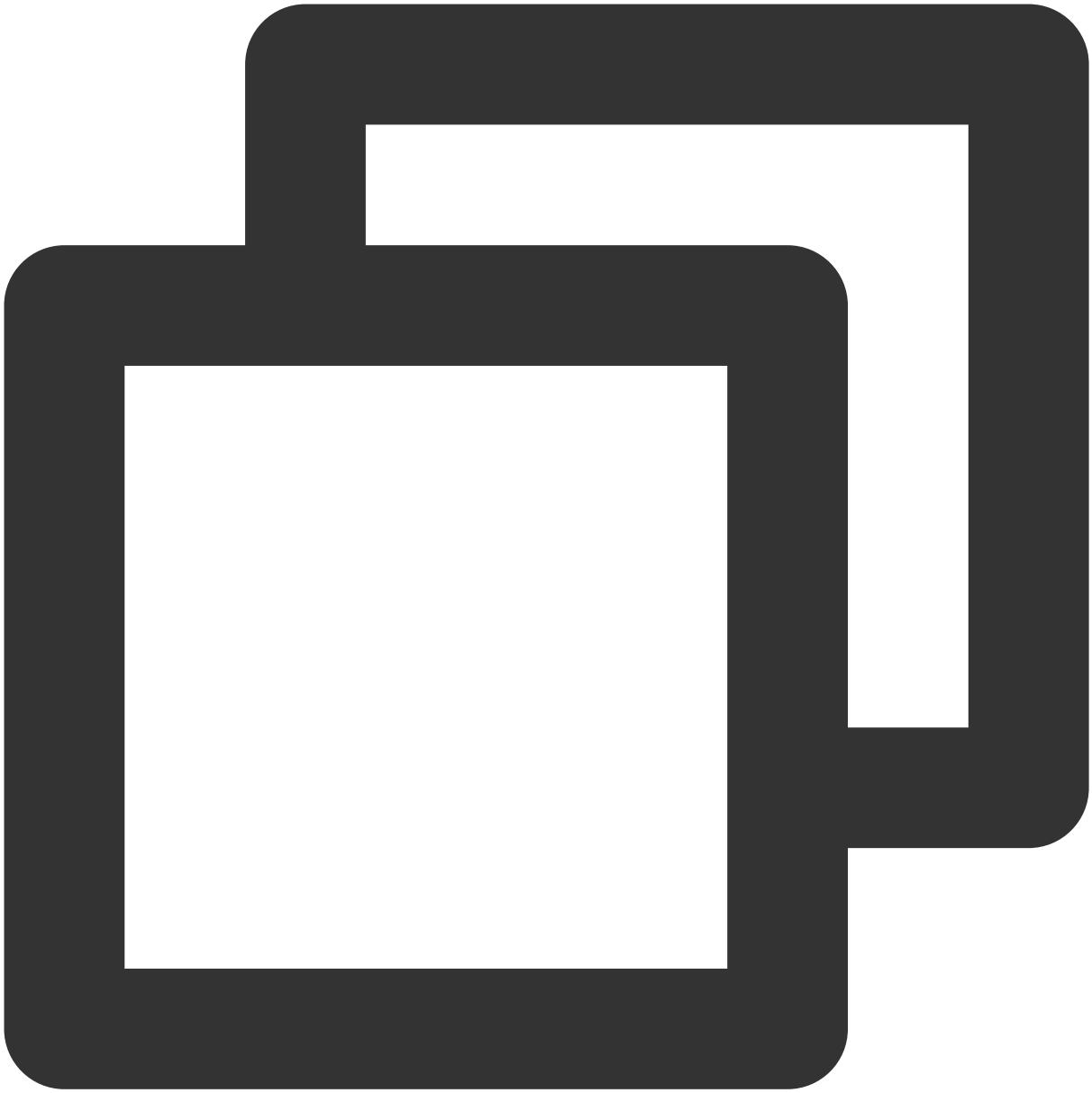
```
config.Producer.Idempotent = true                          //是否需要幂等，在事务场景下需要设
config.Producer.Transaction.Timeout = 1 * time.Minute // 事务超时时间默认1分钟
config.Producer.Transaction.Retry.Max = 50                //事务重试时间
config.Producer.Transaction.Retry.Backoff = 100 * time.Millisecond
```

```
config.Net.MaxOpenRequests = 5                          //默认值5，一次发送请求的数量
config.Producer.Transaction.ID = "test"                 //事务ID
```

需要强调，事务因为要保障消息的 exactly once 语义，因此会额外付出更多的计算资源，所以 config.Net.MaxOpenRequests 的选取必须小于等于5，Broker 端的 ProducerStateManager 实例会缓存每个 PID 在每个 Topic-Partition 上发送的最近 5 个 batch 数据，如果客户在事务的基础上还需要保持一定的吞吐，因此可以设置该值为5，同时适当增加事务超时时间，容忍高负载下一些网络抖动带来的时延问题。

关于压缩参数优化

Sarama Go 支持如下压缩参数：



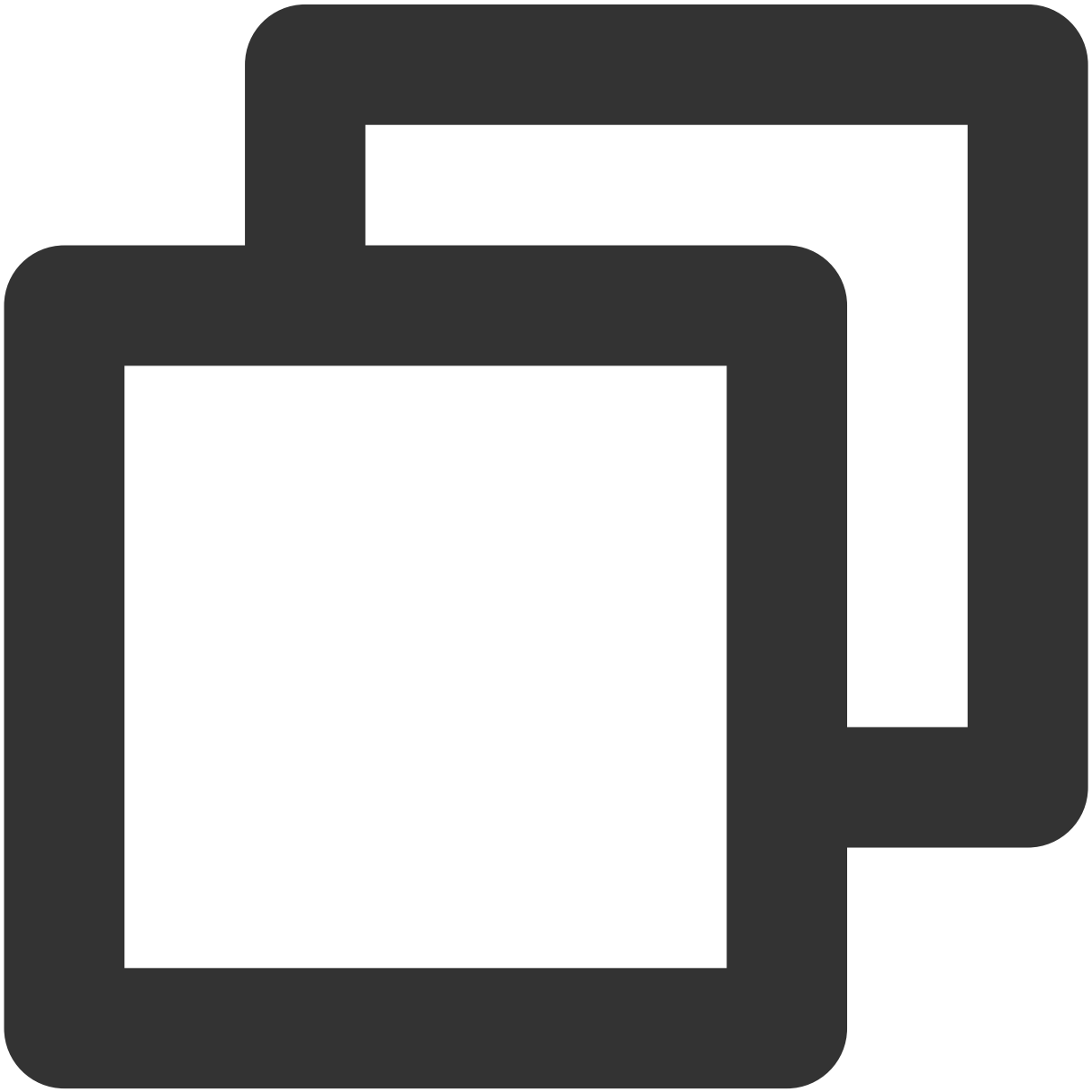
```
config.Producer.Compression = CompressionNone  //对消息是否压缩后发送，默认CompressionN
config.Producer.CompressionLevel = CompressionLevelDefault //指定压缩等级，在配置了压缩
```

在Sarama Kafka Go客户端中，支持以下几种压缩配置：

1. sarama.CompressionNone：不使用压缩。
2. sarama.CompressionGZIP：使用 GZIP 压缩。
3. sarama.CompressionSnappy：使用 Snappy 压缩。
4. sarama.CompressionLZ4：使用 LZ4 压缩。
5. sarama.CompressionZSTD：使用 ZSTD 压缩。

要在 Sarama Kafka Go 客户端中使用压缩消息，需要在创建生产者时设置 config.Producer.Compression 参数。例如，要使用 LZ4 压缩算法，可以将config.Producer.Compression 设置为 sarama.CompressionLZ4 ，虽然压缩消息的压缩和解压缩，发生客户端，是一种用计算换带宽的优化方式，但是由于Broker 针对压缩消息存在校验行为会付出额外的计算成本，尤其是 gzip 压缩，Broker 对其校验计算成本会比较大，在某种程度上可能会出现得不偿失的情况，反而因为计算的增加导致Broker消息处理能力偏低，导致带宽吞吐更低。在低吞吐或者低规格服务下，不建议使用压缩消息。如果还是需要压缩消息，这种情况建议可以使用如下方式进行使用：

1. 在 Producer 端对消息数据独立压缩，生成压缩包数据：messageCompression，同时在消息的 key 存储压缩方式：



```
{"Compression","CompressionLZ4"}
```

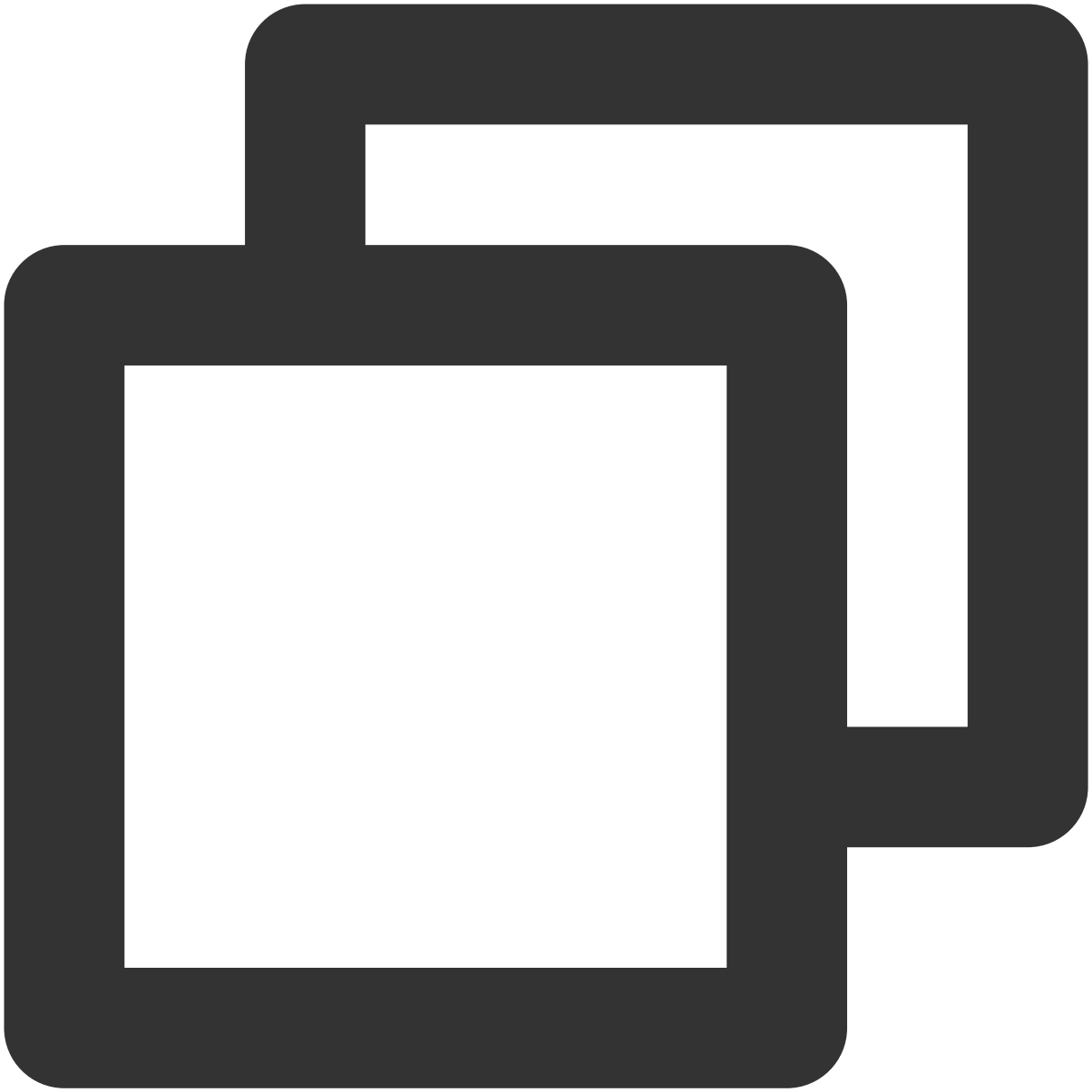
2. 在Producer端将messageCompression当成正常消息发送。
3. 在 Consumer 端读取消息key，获取使用的压缩方式，独立进行解压缩。

创建生产者实例

如果应用程序需要更高的吞吐量，则可以使用异步生产者，以提高消息的发送速度。同时，可以使用批量发送消息的方式，以减少网络开销和 IO 消耗。如果应用程序需要更高的可靠性，则可以使用同步生产者，以确保消息发送成功。同时，可以使用 ACK 确认机制和事务机制，以确保消息的可靠性和一致性。具体的参数调优参考生产者参数与调优。

同步生产者

在 Sarama Kafka Go 客户端中，有两种类型的生产者：同步生产者和异步生产者。它们的主要区别在于发送消息的方式和处理消息结果的方式。同步生产者：同步生产者在发送消息时会阻塞当前线程，直到消息发送完成并收到服务器的确认。因此，同步生产者的吞吐量较低，但是可以立即知道消息是否发送成功。示例如下：

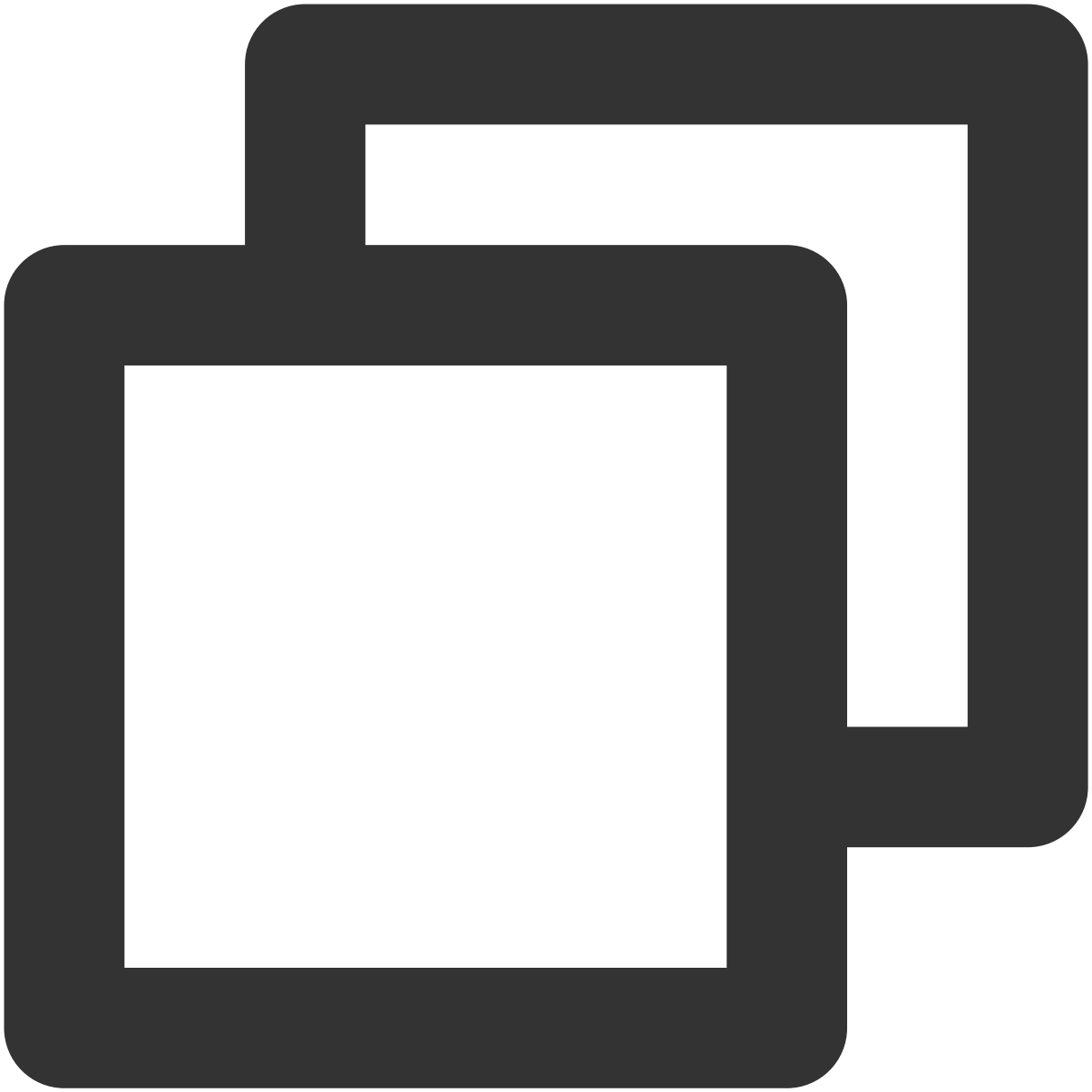


```
package main

import (
    "fmt"
    "log"

    "github.com/Shopify/sarama"
)

func main() {
    config := sarama.NewConfig()
```

```
    config.Producer.RequiredAcks = sarama.WaitForLocal
    config.Producer.Return.Errors = true

    brokers := []string{"localhost:9092"}
    producer, err := sarama.NewSyncProducer(brokers, config)
    if err != nil {
            log.Fatalf("Failed to create producer: %v", err)
    }
    defer producer.Close()

    msg := &sarama.ProducerMessage{
            Topic: "test",
            Value: sarama.StringEncoder("Hello, World!"),
    }

    partition, offset, err := producer.SendMessage(msg)
    if err != nil {
            log.Printf("Failed to send message: %v", err)
    } else {
            fmt.Printf("Message sent to partition %d at offset %d\\n", partitio
    }
}
```

异步生产者

异步生产者：异步生产者在发送消息时不会阻塞当前线程，而是将消息放入一个内部的发送队列，然后立即返回。因此，异步生产者的吞吐量较高，但是需要通过回调函数来处理消息结果。



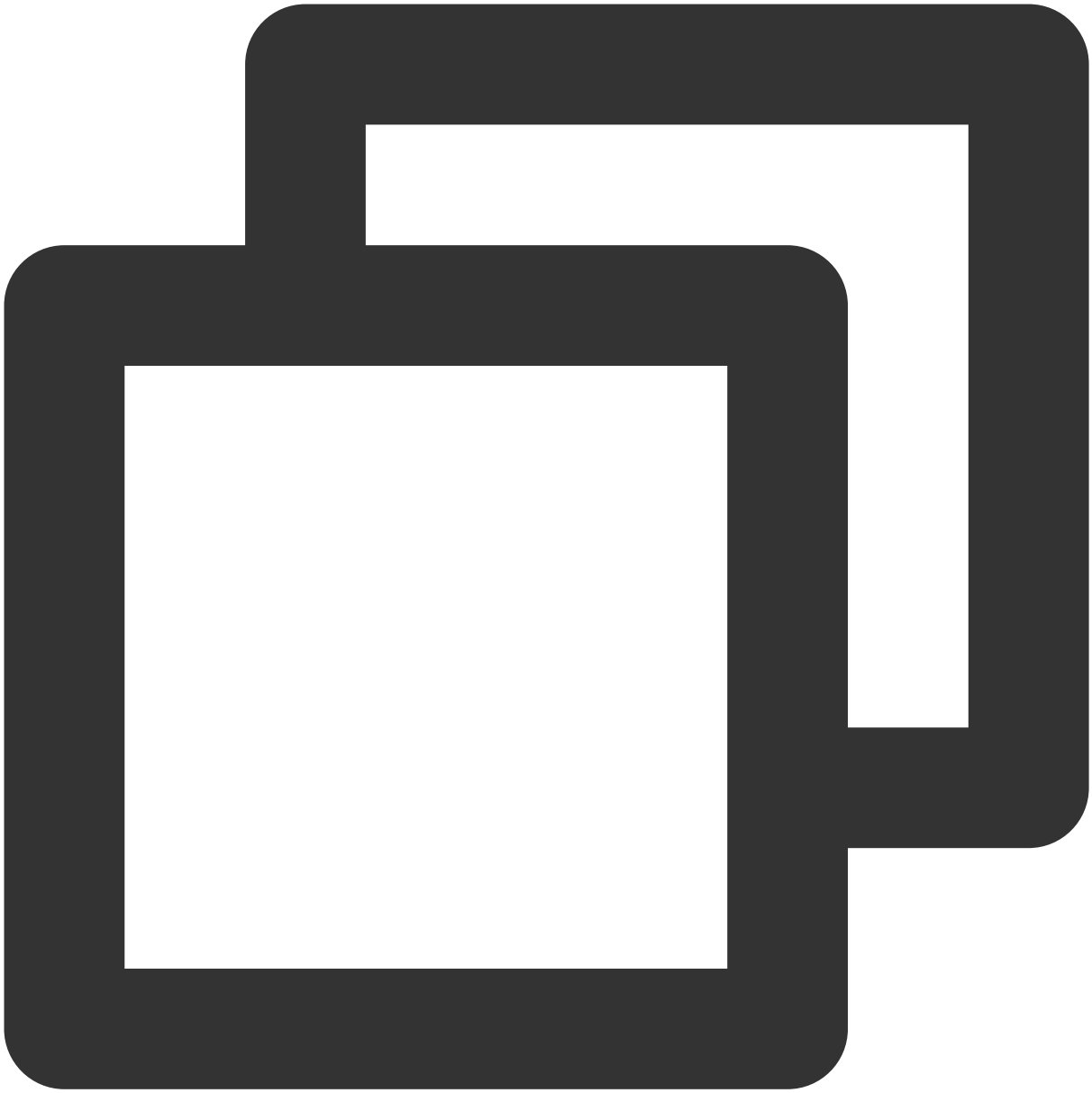
```
package main

import (
    "fmt"
    "log"

    "github.com/Shopify/sarama"
)

func main() {
    config := sarama.NewConfig()
```

```
    config.Producer.RequiredAcks = sarama.WaitForLocal
    config.Producer.Return.Errors = true

    brokers := []string{"localhost:9092"}
    producer, err := sarama.NewAsyncProducer(brokers, config)
    if err != nil {
            log.Fatalf("Failed to create producer: %v", err)
    }
    defer producer.Close()

    msg := &sarama.ProducerMessage{
            Topic: "test",
            Value: sarama.StringEncoder("Hello, World!"),
    }

    producer.Input() <- msg

    select {
    case success := <-producer.Successes():
            fmt.Printf("Message sent to partition %d at offset %d\\n", success.
    case err := <-producer.Errors():
            log.Printf("Failed to send message: %v", err)
    }
}
```

消费者实践

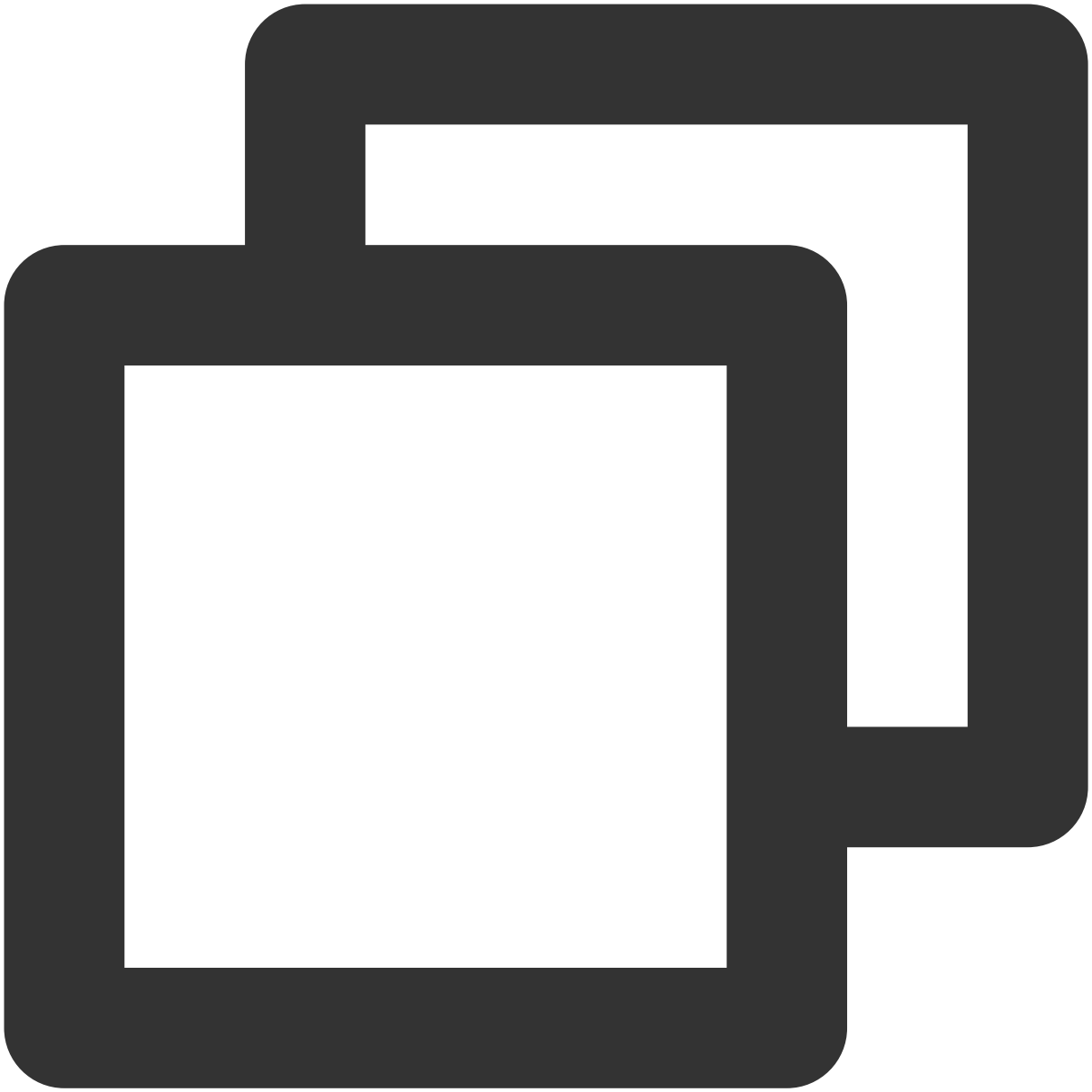
版本选择

在选择 Sarama 客户端版本时，需要确保所选版本与 Kafka broker 版本兼容。Sarama 库支持多个 Kafka 协议版本，可以通过设置 config.Version 来指定使用的协议版本。



```
config := sarama.NewConfig()
config.Version = sarama.V2_8_2_0
```

消费者参数与调优



```
config := sarama.NewConfig()
config.Consumer.Group.Rebalance.Strategy = sarama.NewBalanceStrategyRange //消费者分
config.Consumer.Offsets.Initial = sarama.OffsetNewest    //在没有提交位点情况下，使用最新
config.Consumer.Offsets.AutoCommit.Enable = true       //是否支持自动提交位点，默认支持
config.Consumer.Offsets.AutoCommit.Interval = 1 * time.Second //自动提交位点时间间隔，默
config.Consumer.MaxWaitTime = 250 * time.Millisecond //在没有最新消费消息时候，客户端等待
config.Consumer.MaxProcessingTime = 100 * time.Millisecond
config.Consumer.Fetch.Min = 1                      //消费请求中获取的最小消息字节数,Broke
config.Consumer.Fetch.Max = 0                      //消费请求最大的字节数。默认为0，表示不
config.Consumer.Fetch.Default = 1024 * 1024      //消费请求的默认消息字节数（默认为1MB），
```

```
config.Consumer.Return.Errors = true

config.Consumer.Group.Rebalance.Strategy = sarama.NewBalanceStrategyRange // 设置消费
config.Consumer.Group.Rebalance.Timeout = 60 * time.Second   // 设置rebalance操作的超
config.Consumer.Group.Session.Timeout = 10 * time.Second  // 设置消费者组会话的超时时间
config.Consumer.Group.Heartbeat.Interval = 3 * time.Second  // 心跳超时时间，默认为3s
config.Consumer.MaxProcessingTime = 100 * time.Millisecond //消息处理的超时时间，默认10
```

参数说明与调优

一般消费主要是rebalance时间频繁和消费线程阻塞问题，参考以下说明参数优化：

config.Consumer.Group.Session.Timeout：v0.10.2之前的版本可适当提高该参数值，需要大于消费一批数据的时间，但不要超过30s，建议设置为25s；而v0.10.2及其之后的版本，保持默认值10s 即可。

config.Consumer.Group.Heartbeat.Interval：默认3s，设置该值 需要小于Consumer.Group.Session.Timeout/3。

config.Consumer.Group.Rebalance.Timeout：默认60s，如果分区数和消费者较多，建议适当调大该值。

config.Consumer.MaxProcessingTime：该值要大于<max.poll.records> / (<单个线程每秒消费的条数> * <消费线程的个数>)的值。

注意：

根据需求调大 MaxProcessingTime 时间。

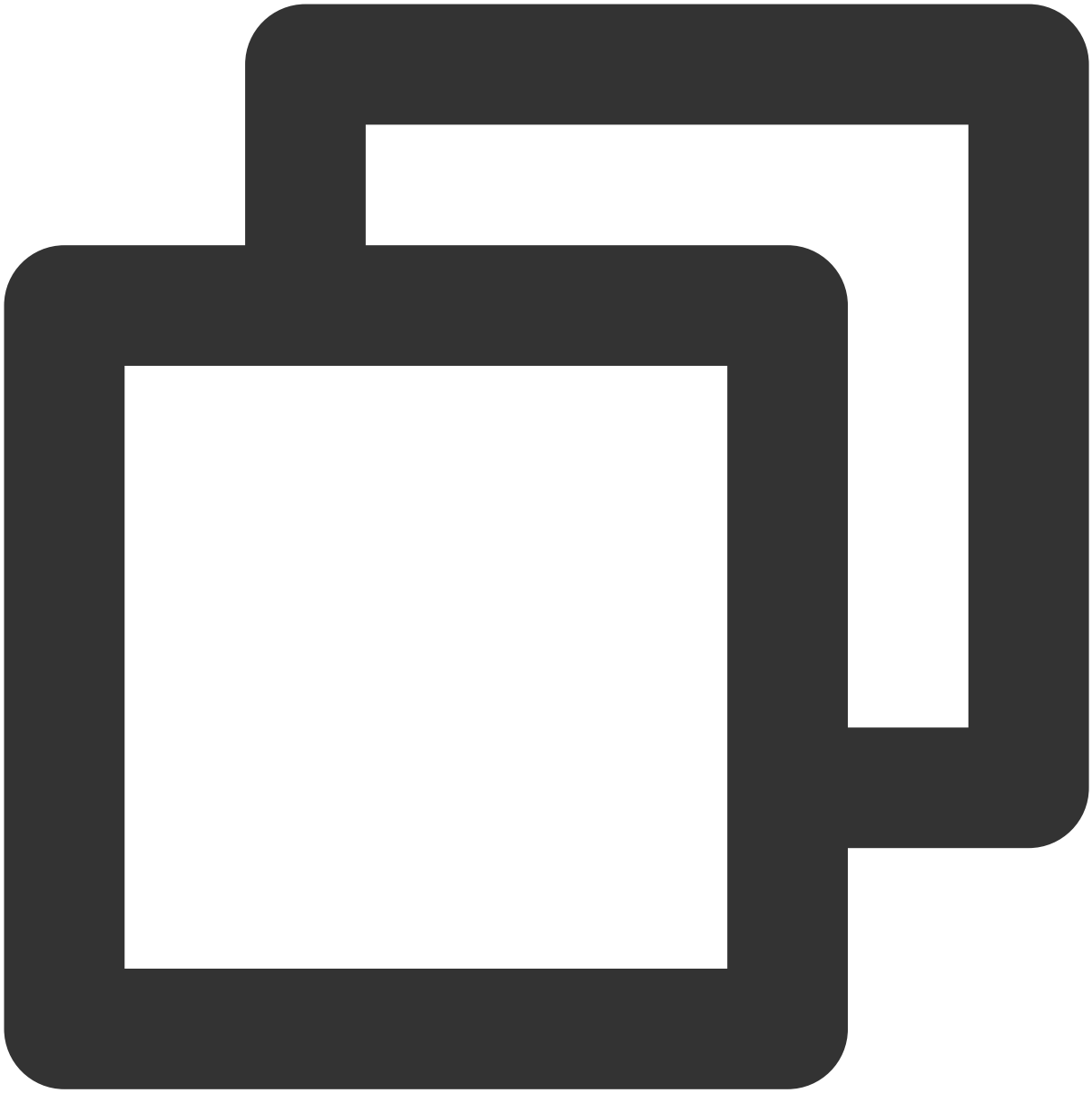
针对处理时间大于 MaxProcessingTime 请求处理时间进行监控，采样打印超时时间。

创建消费者实例

Sarama 提供订阅的模型创建消费者，其中在提交位点方面，提供手动提交位点和自动提交位点两种方式。

自动提交位点

自动提交位点：消费者在拉取消息后会自动提交位点，无需手动操作。这种方式的优点是简单易用，但是可能会导致消息重复消费或丢失。



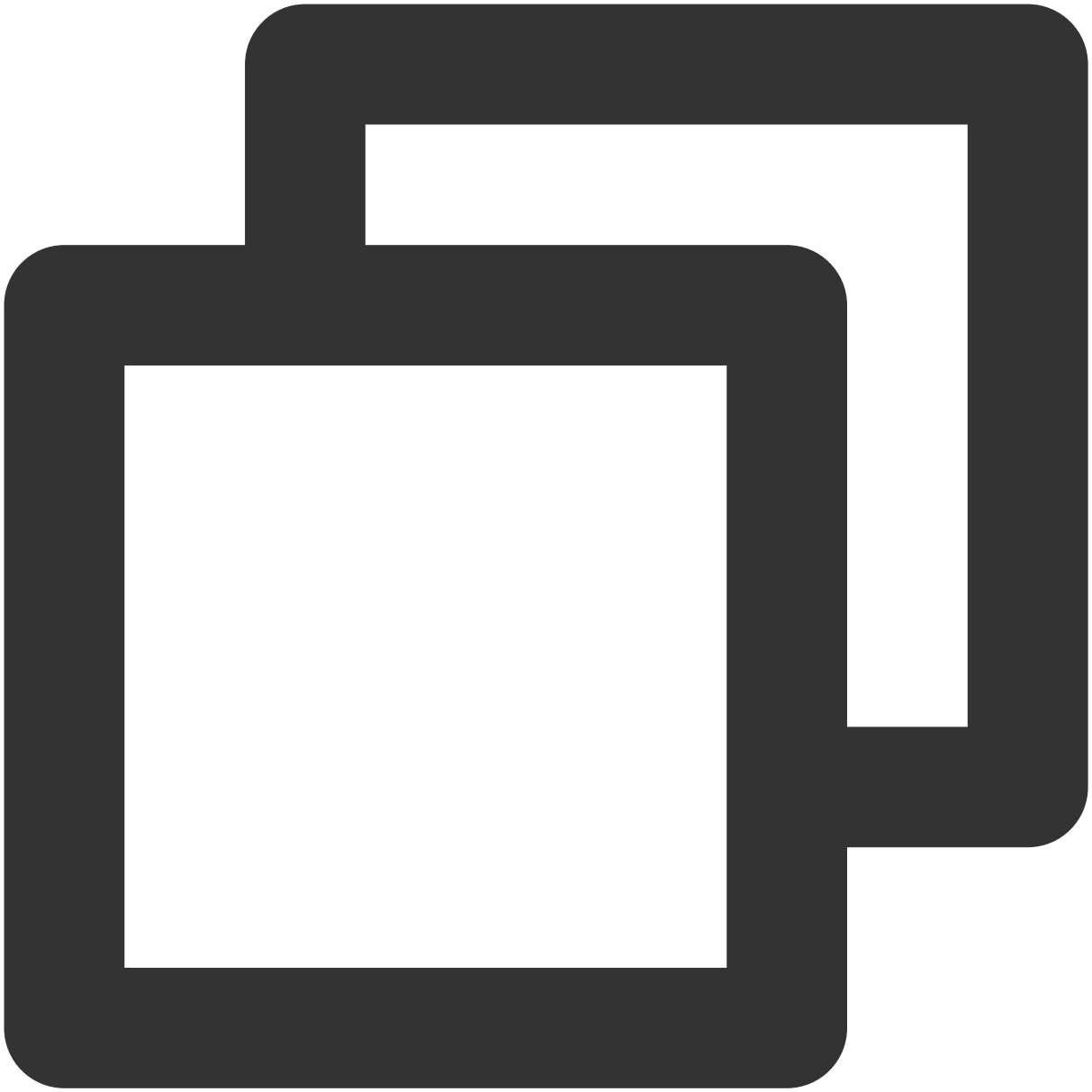
```
package main

import (
    "context"
    "fmt"
    "log"
    "os"
    "os/signal"
    "sync"
    "time"
```

```
    "github.com/Shopify/sarama"
)

func main() {
    config := sarama.NewConfig()
    config.Version = sarama.V2_1_0_0
    config.Consumer.Offsets.Initial = sarama.OffsetOldest
    config.Consumer.Offsets.AutoCommit.Enable = true
    config.Consumer.Offsets.AutoCommit.Interval = 1 * time.Second

    brokers := []string{"localhost:9092"}
    topic := "test-topic"

    client, err := sarama.NewConsumerGroup(brokers, "test-group", config)
    if err != nil {
            log.Fatalf("unable to create kafka consumer group: %v", err)
    }
    defer client.Close()

    ctx, cancel := context.WithCancel(context.Background())
    signals := make(chan os.Signal, 1)
    signal.Notify(signals, os.Interrupt)

    var wg sync.WaitGroup
    wg.Add(1)

    go func() {
            defer wg.Done()

            for {
                    err := client.Consume(ctx, []string{topic}, &consumerHandle
                    if err != nil {
                            log.Printf("consume error: %v", err)
                    }

                    select {
                    case <-signals:
                            cancel()
                            return
                    default:
                    }
            }
    }()

    wg.Wait()
}
```

```
type consumerHandler struct{}

func (h *consumerHandler) Setup(sarama.ConsumerGroupSession) error {
    return nil
}

func (h *consumerHandler) Cleanup(sarama.ConsumerGroupSession) error {
    return nil
}

func (h *consumerHandler) ConsumeClaim(sess sarama.ConsumerGroupSession, claim sara
    for msg := range claim.Messages() {
                fmt.Printf("Received message: key=%s, value=%s, partition=%d, offse
                sess.MarkMessage(msg, "")
    }
    return nil
}
```

手动提交位点

手动提交位点：消费者在处理完消息后需要手动提交位点。这种方式的优点是可以精确控制位点的提交，避免消息重复消费或丢失。但是需要注意，手动提交位点如果太频繁会导致 Broker CPU 很高，影响性能，随着消息量增加，CPU 消费会很高，影响正常 Broker 的其他功能，因此建议间隔一定消息提交位点。



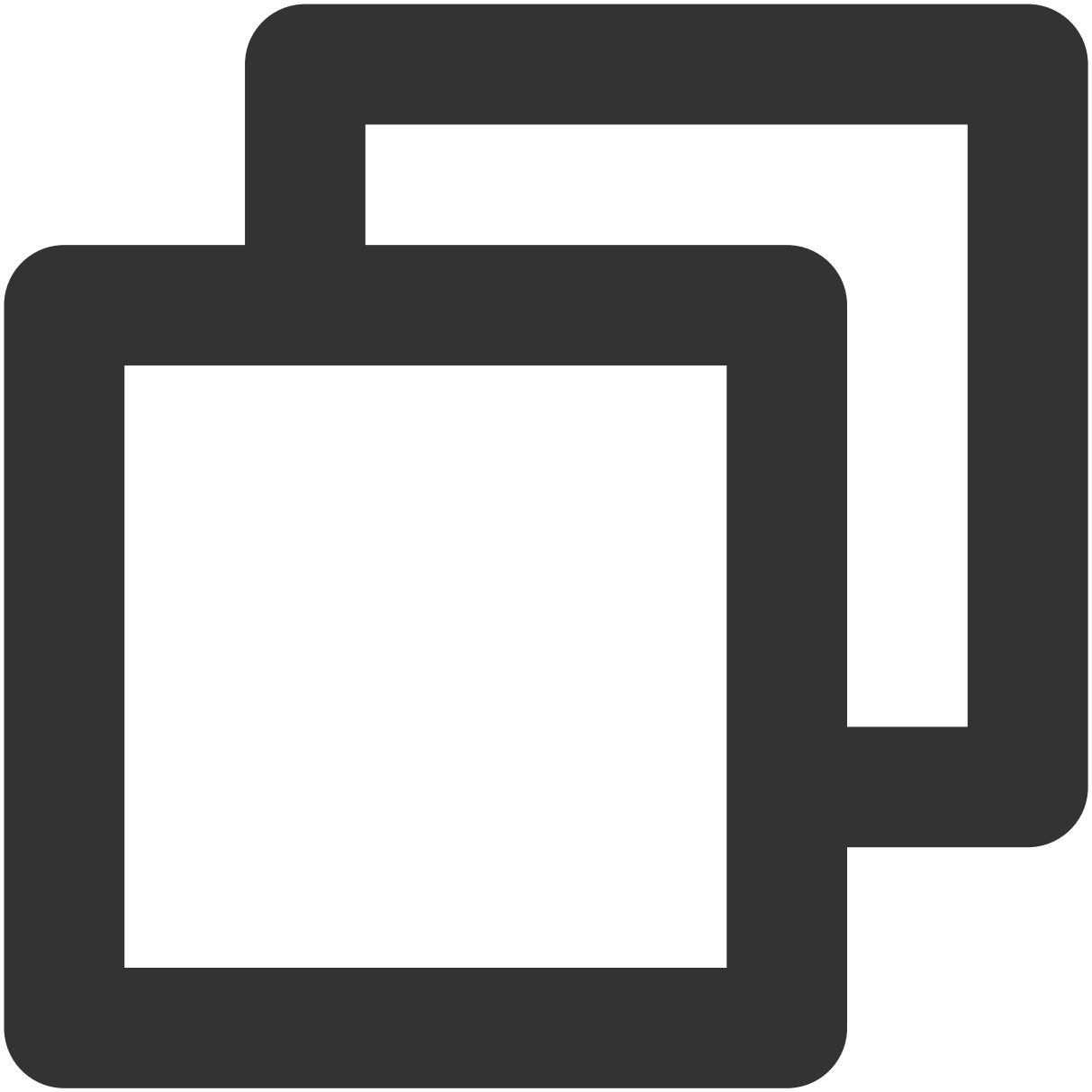
```
package main

import (
    "context"
    "fmt"
    "log"
    "os"
    "os/signal"
    "sync"

    "github.com/Shopify/sarama"
```

```
)

func main() {
    config := sarama.NewConfig()
    config.Version = sarama.V2_1_0_0
    config.Consumer.Offsets.Initial = sarama.OffsetOldest
    config.Consumer.Offsets.AutoCommit.Enable = false

    brokers := []string{"localhost:9092"}
    topic := "test-topic"

    client, err := sarama.NewConsumerGroup(brokers, "test-group", config)
    if err != nil {
            log.Fatalf("unable to create kafka consumer group: %v", err)
    }
    defer client.Close()

    ctx, cancel := context.WithCancel(context.Background())
    signals := make(chan os.Signal, 1)
    signal.Notify(signals, os.Interrupt)

    var wg sync.WaitGroup
    wg.Add(1)

    go func() {
            defer wg.Done()

            for {
                    err := client.Consume(ctx, []string{topic}, &consumerHandle
                    if err != nil {
                            log.Printf("consume error: %v", err)
                    }

                    select {
                    case <-signals:
                            cancel()
                            return
                    default:
                    }
            }
    }()

    wg.Wait()
}

type consumerHandler struct{}
```

```
func (h *consumerHandler) Setup(sarama.ConsumerGroupSession) error {
    return nil
}

func (h *consumerHandler) Cleanup(sarama.ConsumerGroupSession) error {
    return nil
}

func (h *consumerHandler) ConsumeClaim(sess sarama.ConsumerGroupSession, claim sara
    for msg := range claim.Messages() {
                fmt.Printf("Received message: key=%s, value=%s, partition=%d, offse
                sess.MarkMessage(msg, "")
        sess.Commit()
    }
    return nil
}
```

Sarama Go 生产消费常见问题

1. 配置了手动提交位点，但是位点在控制台查询消费组时候没有原因。
无论配置了手动提交位点还是自动提交位点，都需要先进行标记，sess.MarkMessage(msg, "")，表示该消息已经被消费完，然后才能提交位点。
2. Sarama Go 作为消费者的一些问题，Sarama Go 版本客户端存在以下已知问题：
2.1 当Topic新增分区时，Sarama Go客户端无法感知并消费新增分区，需要客户端重启后，才能消费到新增分区。
2.2 当Sarama Go客户端同时订阅两个以上的Topic时，有可能会导致部分分区无法正常消费消息。
2.3 当Sarama Go客户端的消费位点重置策略设置为Oldest(earliest)时，如果客户端宕机或服务端版本升级，由于Sarama Go客户端自行实现OutOfRange机制，有可能会导致客户端从最小位点开始重新消费所有消息。
2.4 对于该问题：Confluent Go客户端的Demo地址，请访问 kafka-confluent-go-demo。
3. 出现报错：Failed to produce message to topic。
原因可能为版本没有对齐，此时客户先确定kafka Broker的版本，然后指定版本：



```
config := sarama.NewConfig()
config.Version = sarama.V2_1_0_0
```

Java SDK

最近更新时间：2024-07-04 16:00:59

背景

TDMQ CKafka 是一个分布式流处理平台，用于构建实时数据管道和流式应用程序。它提供了高吞吐量、低延迟、可伸缩性和容错性等特性。
Kafka Clients：Kafka 自带的客户端，通过 Java 实现，是 Kafka 生产和消费标准协议的客户端。
本文着重介绍上述 Java 客户端的关键参数，实践教程以及常见问题。

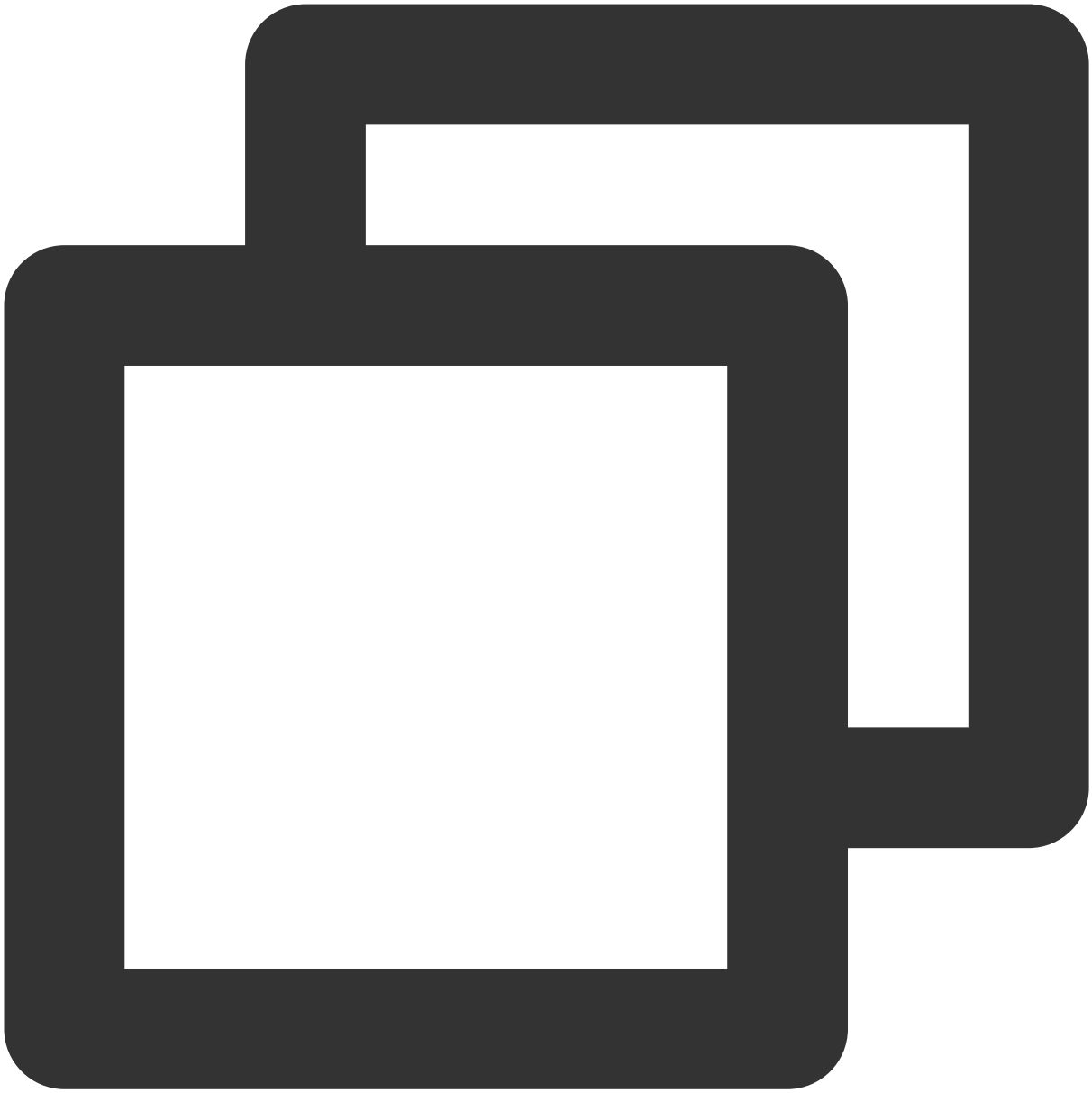
生产者实践

版本选择

Kafka 客户端和集群之间的兼容性非常重要，通常情况下，较新版本的客户端可以兼容较旧版本的集群，但反之则不一定成立。一般情况下，CKafka实例 Broker 在部署后是明确的，因此可以直接根据 Broker 的版本选择相匹配的客户端的版本。
Java 生态中，广泛使用 Spring Kafka，其中 Spring Kafka 版本和 Kafka Broker 版本的对应关系，可以参见 Spring 官方网址的版本对应关系。
生产者参数与调优

生产者参数

在使用 Kafka Client 客户端写入 Kafka 时候，需要配置如下关键参数，相关的参数和默认值如下：



```
import org.apache.kafka.clients.producer.*;
import org.apache.kafka.common.serialization.StringSerializer;

import java.util.Properties;
import java.util.concurrent.ExecutionException;

public class KafkaProducerExample {

    private static final String BOOTSTRAP_SERVERS = "localhost:9092";
    private static final String TOPIC = "test-topic";
```

```
public static void main(String[] args) throws ExecutionException, InterruptedEx
    // 创建Kafka生产者配置
    Properties props = new Properties();
    props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS); //Ka
    props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.clas
    props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.cl

    // 设置生产者关键参数以及默认值
    props.put(ProducerConfig.ACKS_CONFIG, "1");//acks,默认值为1,消息确认的级别。0表
    props.put(ProducerConfig.BATCH_SIZE_CONFIG, 16384);//batch.size,批量发送的大小
    props.put(ProducerConfig.BUFFER_MEMORY_CONFIG, 33554432);//buffer.memory,生
    props.put(ProducerConfig.CLIENT_ID_CONFIG, "");//client.id,客户端ID。这个ID可
    props.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "none");//compression.typ
    props.put(ProducerConfig.CONNECTIONS_MAX_IDLE_MS_CONFIG, 540000);//connecti
    props.put(ProducerConfig.DELIVERY_TIMEOUT_MS_CONFIG, 120000);//delivery.tim
    props.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, false);//enable.idempot
    props.put(ProducerConfig.INTERCEPTOR_CLASSES_CONFIG, "");//interceptor.clas
    props.put(ProducerConfig.LINGER_MS_CONFIG, 0);//linger.ms延迟发送的时间，单位为
    props.put(ProducerConfig.MAX_BLOCK_MS_CONFIG, 60000);//max.block.ms,生产者在
    props.put(ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION, 5);//max.in
    props.put(ProducerConfig.MAX_REQUEST_SIZE_CONFIG, 1048576);//max.request.si
    props.put(ProducerConfig.METADATA_MAX_AGE_CONFIG, 300000);//metadata.max.ag
    props.put(ProducerConfig.METRIC_REPORTER_CLASSES_CONFIG, "");//metric.repor
    props.put(ProducerConfig.PARTITIONER_CLASS_CONFIG, "org.apache.kafka.client
    props.put(ProducerConfig.RECEIVE_BUFFER_CONFIG, 32768);//receive.buffer.byt
    props.put(ProducerConfig.SEND_BUFFER_CONFIG, 131072);//send.buffer.bytes发送
    props.put(ProducerConfig.RECONNECT_BACKOFF_MAX_MS_CONFIG, 1000);//reconnect
    props.put(ProducerConfig.RECONNECT_BACKOFF_MS_CONFIG, 50);//reconnect.backo
    props.put(ProducerConfig.REQUEST_TIMEOUT_MS_CONFIG, 30000);//request.timeou
    props.put(ProducerConfig.RETRIES_CONFIG, 2147483647);//retries发送失败时的重试
    props.put(ProducerConfig.RETRY_BACKOFF_MS_CONFIG, 100);//retry.backoff.ms重
    props.put(ProducerConfig.TRANSACTION_TIMEOUT_CONFIG, 60000);//transaction.t
    props.put(ProducerConfig.TRANSACTIONAL_ID_CONFIG, null);//transactional.id事
    props.put(ProducerConfig.CLIENT_DNS_LOOKUP_CONFIG, "default");//client.dns.

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

    // 发送消息
    for (int i = 0; i < 100; i++) {
        String key = "key-" + i;
        String value = "value-" + i;

        // 创建消息记录
        ProducerRecord<String, String> record = new ProducerRecord<>(TOPIC, key

        // 发送消息
```

```
            producer.send(record, new Callback() {
                @Override
                public void onCompletion(RecordMetadata metadata, Exception excepti
                    if (exception == null) {
                        System.out.println("消息发送成功：key=" + key + ", value=" + v
                    } else {
                        System.err.println("消息发送失败：" + exception.getMessage())
                    }
                }
            });
        }

        // 关闭生产者
        producer.close();
    }
}
```

参数说明调优

关于 acks 参数优化

acks 参数用于控制生产者发送消息时的确认机制。该参数的默认值为1，表示消息发送给 Leader Broker 后，Leader 确认消息写入后即返回。acks参数还有以下可选值：

0: 不等待任何确认，直接返回。

1: 等待 Leader 副本确认写入后返回。

-1或者 all: 等待 Leader 副本以及相关的 Follower 副本确认写入后返回。

由上可知，在跨可用区场景，以及副本数较多的 Topic，acks 参数的取值会影响消息的可靠性和吞吐量。因此：

在一些在线业务消息的场景下，吞吐量要求不大，可以将 acks 参数设置为-1，确保消息被所有副本接收和确认后才返回，从而提高消息的可靠性，但是会牺牲写入吞吐和性能，时延会增加。

在日志采集等大数据或者离线计算的场景下，要求高吞吐（即每秒写入 Kafka 的数据量）的情况下，可以将 acks 设置为1，提高吞吐量。

关于 Batch 相关参数优化

默认情况下，传输同等数据量的情况下，多次请求和一次请求的网络传输，一次请求传输能有效减少相关计算和网络资源，提高整体写入的吞吐量。

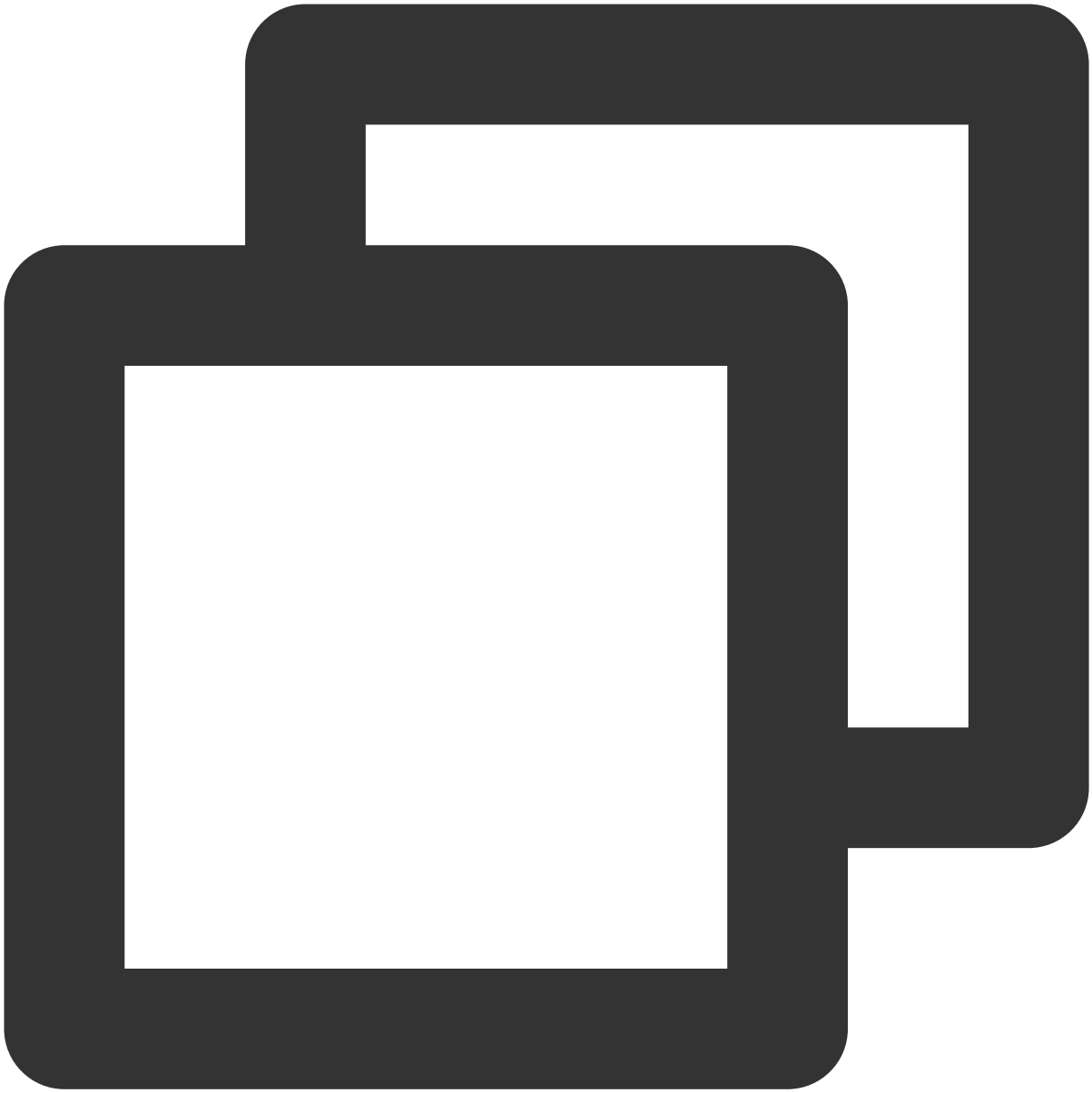
因此，可以通过这个参数设置优化客户端发送消息的吞吐能力。在高吞吐场景下，可以配合计算和设置：

batch.size：默认16K。

linger.ms：默认为0，可以适当增加耗时，如设置100ms，尽可能聚合更多消息批量发送消息。

buffer.memory：默认32MB，对于大流量 Producer，在堆内存充足情况可以设置更大，如设置256MB。

关于事务参数优化

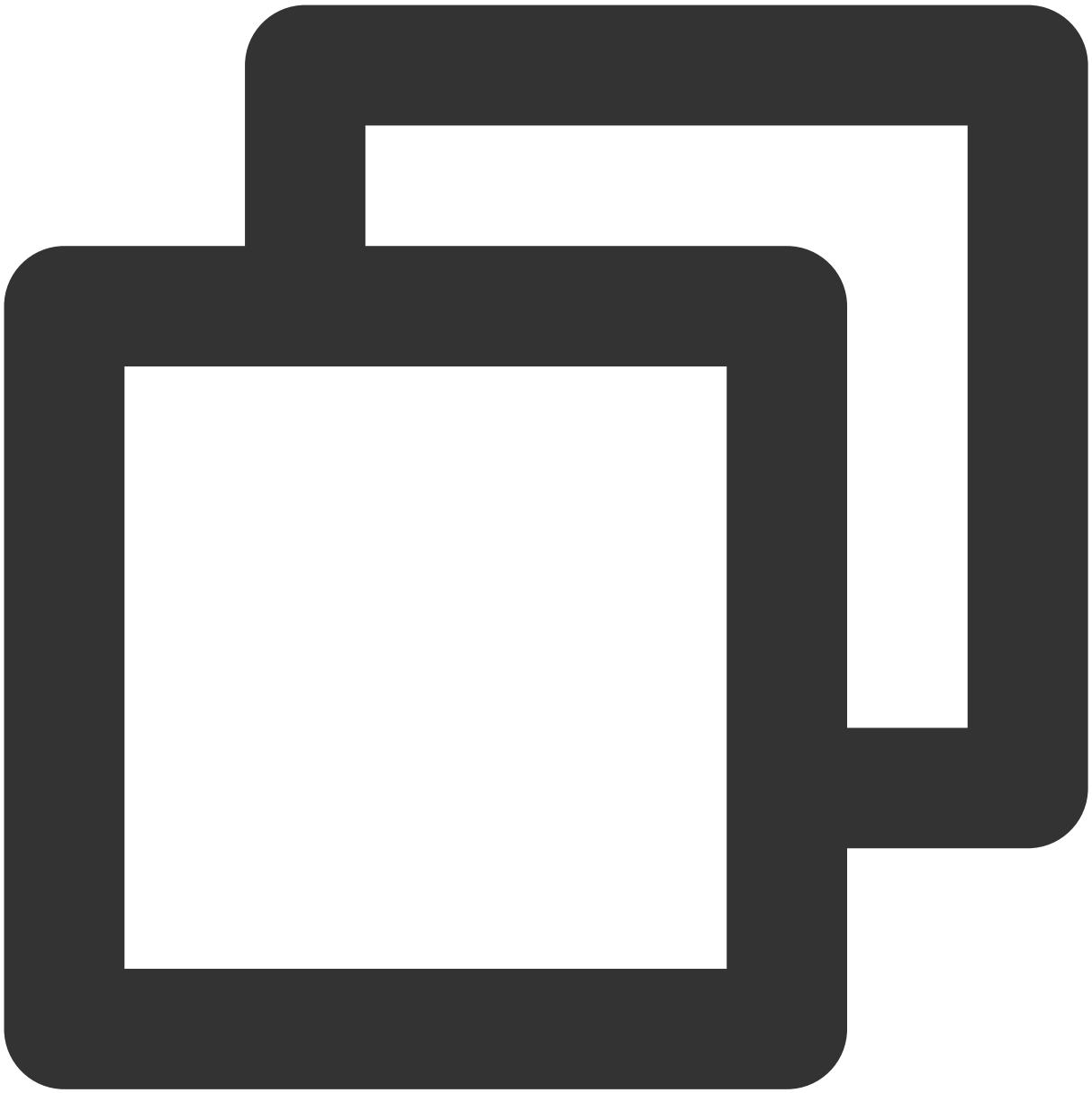


```
props.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, false);//enable.idempotence,是否
props.put(ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION, 5);//max.in.flight.
props.put(ProducerConfig.TRANSACTIONAL_ID_CONFIG, null);//transactional.id事务ID。如
props.put(ProducerConfig.TRANSACTION_TIMEOUT_CONFIG, 60000);//transaction.timeout.m
```

需要强调，事务因为要保障消息的 exactly once 语义，因此会额外付出更多的计算资源。
对于事务场景，可是适当增加事务超时时间，容忍高吞吐场景下，写入延时带来的抖动。

关于压缩参数优化

Kafka Java Client 支持如下压缩参数：



```
props.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "none");//compression.type消息压缩
```

目前支持以下几种压缩配置：

none：不使用压缩算法。

gzip：使用 GZIP 压缩算法。

snappy：使用 Snappy 压缩算法。

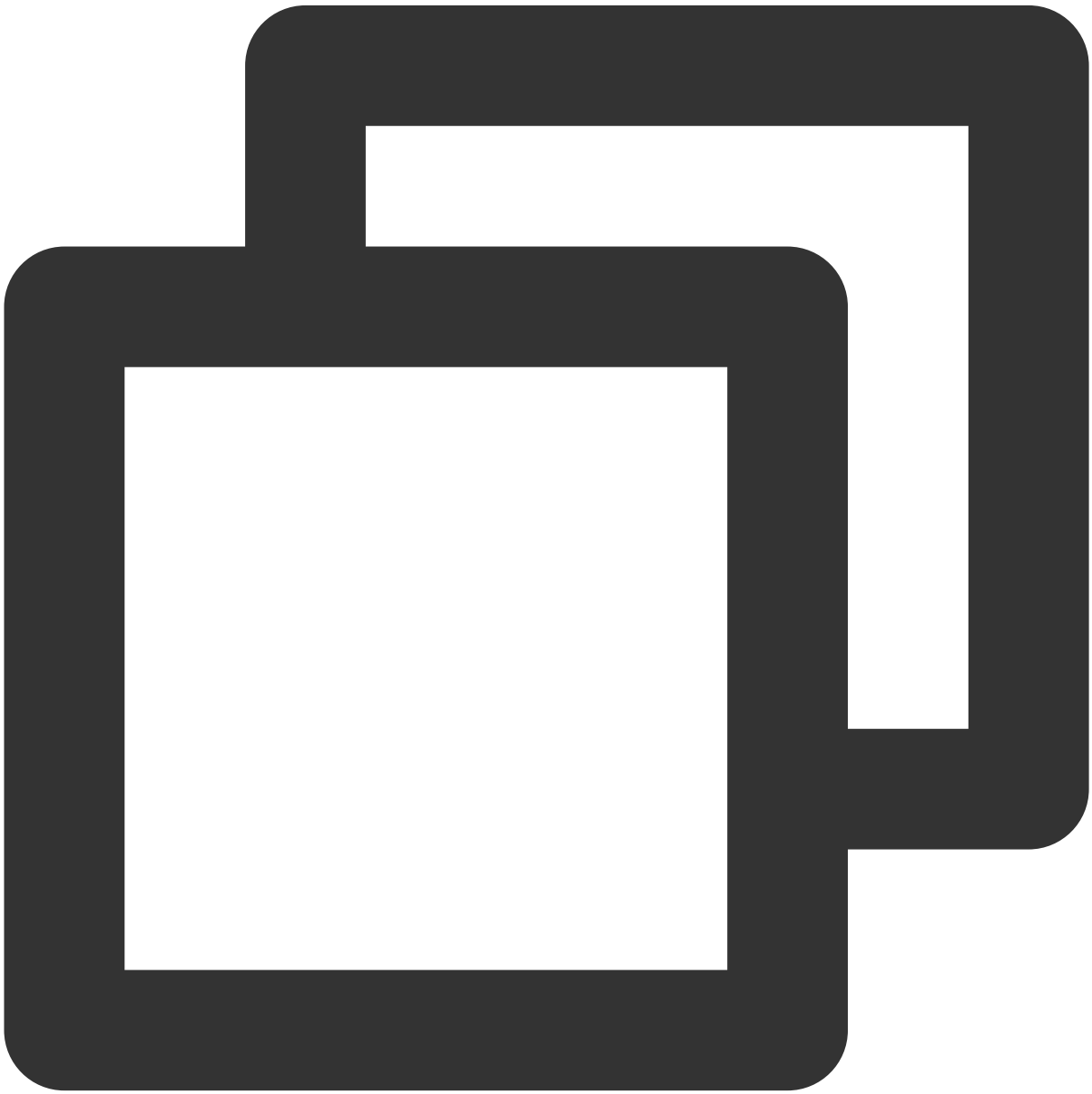
lz4：使用 LZ4 压缩算法。

zstd：使用 ZSTD 压缩算法。

要在 Kafka Java 客户端中使用压缩消息，需要在创建生产者时设置 compression.type 参数。例如，要使用 LZ4 压缩算法，可以将compression.type 设置为lz4。

Kafka 压缩消息是一种用计算换带宽的优化方式，虽然 Kafka 压缩消息的压缩和解压缩，发生在客户端，但是由于 Broker 针对压缩消息存在校验行为会付出额外的计算成本，尤其是 gzip 压缩，服务端对该压缩消息校验的计算成本会非常大，在某种程度上可能会出现得不偿失的情况，因为计算的增加导致 Broker CPU 利用率很高，降低了其他请求的处理能力，导致整体性能更低。这种情况建议可以使用如下方式规避 Broker 的校验：

1. 在 Producer 端对消息数据独立压缩，生成压缩包数据：messageCompression，同时在消息的 key 存储压缩方式：



```
{"Compression","lz4"}
```

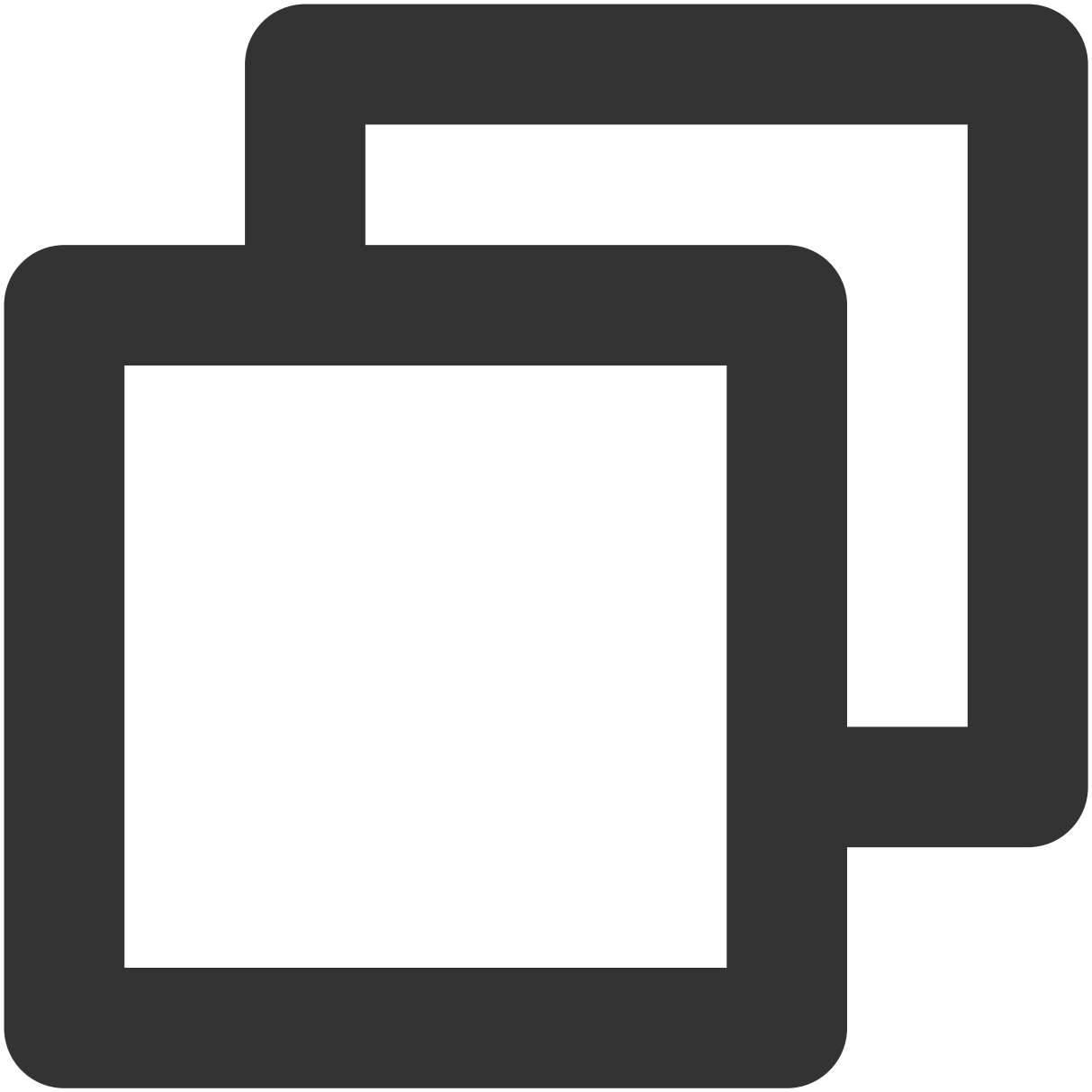
2. 在 Producer 端将 messageCompression 当成正常消息发送。
3. 在 Consumer 段读取消息 key，获取使用的压缩方式，独立进行解压缩。

创建生产者实例

如果应用程序需要更高的吞吐量，可以使用异步发送，以提高消息的发送速度。同时，可以使用批量发送消息的方式，以减少网络开销和 IO 消耗。如果应用程序需要更高的可靠性，可以使用同步发送，以确保消息发送成功。同时，可以使用 ACK 确认机制和事务机制，以确保消息的可靠性和一致性。具体的参数调优参考生产者参数与调优。

同步发送

在 Kafka Java Client 客户端中，同步发送的示例代码如下:

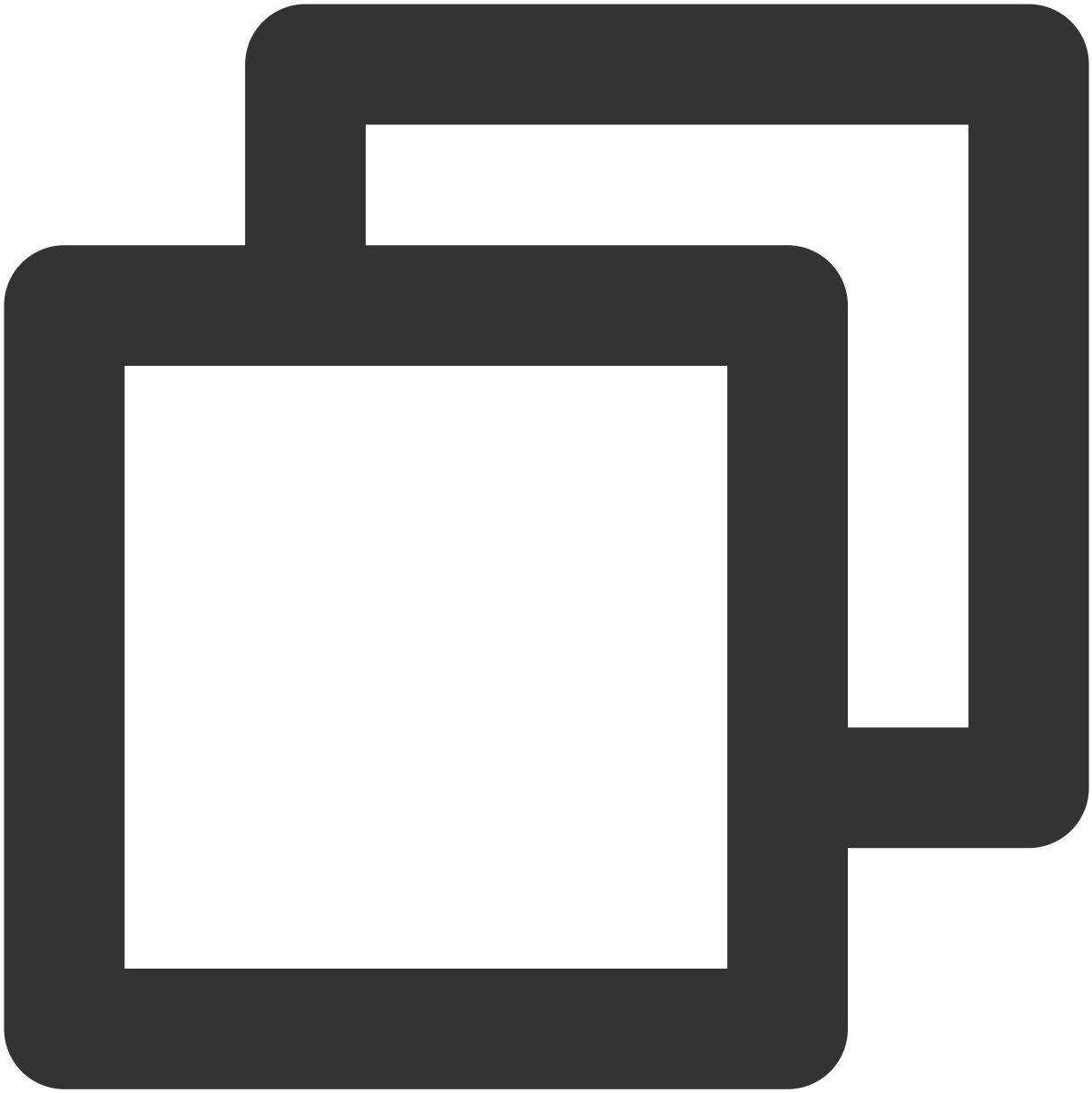


```
import org.apache.kafka.clients.producer.*;
import org.apache.kafka.common.serialization.StringSerializer;

import java.util.Properties;
import java.util.concurrent.ExecutionException;

public class KafkaProducerSyncExample {

    private static final String BOOTSTRAP_SERVERS = "localhost:9092";
    private static final String TOPIC = "test-topic";
```

```
    public static void main(String[] args) {
        // 创建Kafka生产者配置
        Properties props = new Properties();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS);
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.clas
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.cl

        // 设置生产者参数
        props.put(ProducerConfig.ACKS_CONFIG, "all");
        props.put(ProducerConfig.RETRIES_CONFIG, 3);

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

        // 同步发送消息
        for (int i = 0; i < 10; i++) {
            String key = "sync-key-" + i;
            String value = "sync-value-" + i;

            // 创建消息记录
            ProducerRecord<String, String> record = new ProducerRecord<>(TOPIC, key

            try {
                // 发送消息并等待结果
                RecordMetadata metadata = producer.send(record).get();
                System.out.println("同步发送成功：key=" + key + ", value=" + value);
            } catch (InterruptedException | ExecutionException e) {
                System.err.println("同步发送失败：" + e.getMessage());
            }
        }

        // 关闭生产者
        producer.close();
    }
}
```

异步发送

异步发送：异步发送消息时不会阻塞当前线程，生产者的吞吐量较高，但是需要通过回调函数来处理消息结果，示例如下：

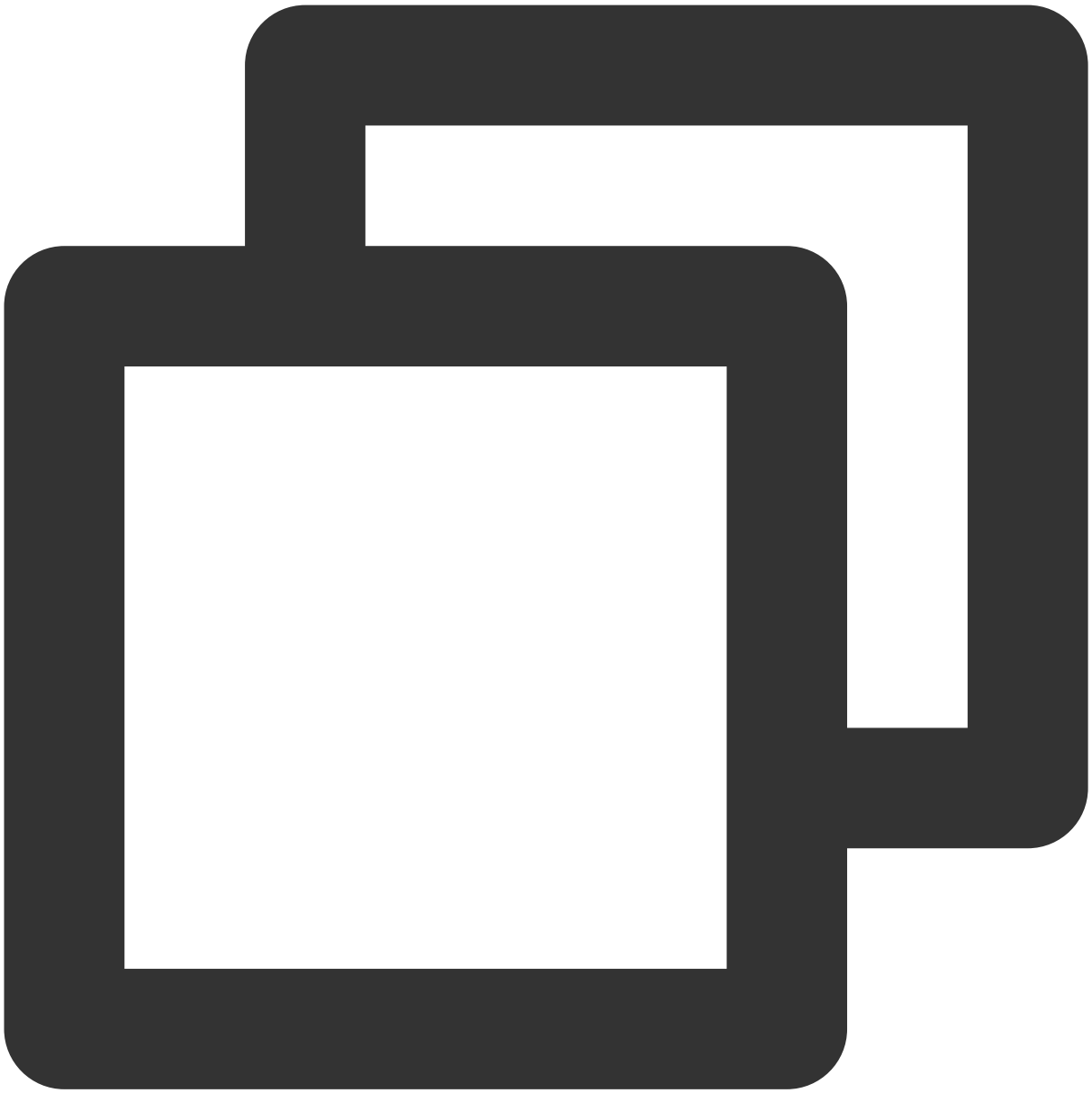


```
import org.apache.kafka.clients.producer.*;
import org.apache.kafka.common.serialization.StringSerializer;

import java.util.Properties;

public class KafkaProducerAsyncExample {

    private static final String BOOTSTRAP_SERVERS = "localhost:9092";
    private static final String TOPIC = "test-topic";
```

```
    public static void main(String[] args) {
        // 创建Kafka生产者配置
        Properties props = new Properties();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS);
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.clas
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.cl

        // 设置生产者参数
        props.put(ProducerConfig.ACKS_CONFIG, "all");
        props.put(ProducerConfig.RETRIES_CONFIG, 3);

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

        // 异步发送消息
        for (int i = 0; i < 10; i++) {
            String key = "async-key-" + i;
            String value = "async-value-" + i;

            // 创建消息记录
            ProducerRecord<String, String> record = new ProducerRecord<>(TOPIC, key

            // 发送消息并设置回调函数
            producer.send(record, new Callback() {
                @Override
                public void onCompletion(RecordMetadata metadata, Exception excepti
                    if (exception == null) {
                        System.out.println("异步发送成功：key=" + key + ", value=" + v
                    } else {
                        System.err.println("异步发送失败：" + exception.getMessage());
                    }
                }
            });
        }

        // 关闭生产者
        producer.close();
    }
}
```

消费者实践

消费者参数



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

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

```
import java.util.Properties;

public class KafkaConsumerDemo {

    public static void main(String[] args) {
        Properties properties = new Properties();
        properties.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        properties.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserial
        properties.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeseri
        properties.put(ConsumerConfig.GROUP_ID_CONFIG, "test-group"); // "group.id"
        properties.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "latest"); // "auto
        properties.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "true"); // "enabl
        properties.put(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CONFIG, "5000"); // "
        properties.put(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG, "10000"); // "sess
        properties.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, "500"); // "max.poll
        properties.put(ConsumerConfig.MAX_POLL_INTERVAL_MS_CONFIG, "300000"); // "m
        properties.put(ConsumerConfig.FETCH_MIN_BYTES_CONFIG, "1"); // "fetch.min.b
        properties.put(ConsumerConfig.FETCH_MAX_BYTES_CONFIG, "52428800"); // "fetc
        properties.put(ConsumerConfig.FETCH_MAX_WAIT_MS_CONFIG, "500"); // "fetch.m
        properties.put(ConsumerConfig.HEARTBEAT_INTERVAL_MS_CONFIG, "3000"); // "he
        properties.put(ConsumerConfig.CLIENT_ID_CONFIG, "my-client-id"); // "client
        properties.put(ConsumerConfig.REQUEST_TIMEOUT_MS_CONFIG, "30000"); // "requ

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

        try {
            while (true) {
                ConsumerRecords<String, String> records = consumer.poll(Duration.of
                for (ConsumerRecord<String, String> record : records) {
                    System.out.printf("offset = %d, key = %s, value = %s%n", record
                }
            }
        } finally {
            consumer.close();
        }
    }
}
```

参数调优

1. 在使用 Kafka 消费者时，我们可以通过调整一些参数来优化性能。以下是一些常见的参数调优方案：

fetch.min.bytes：如果不确定消息最低大小，这个参数建议设置为1，如果明确消息最小值，可以设置该值为最小消息大小。

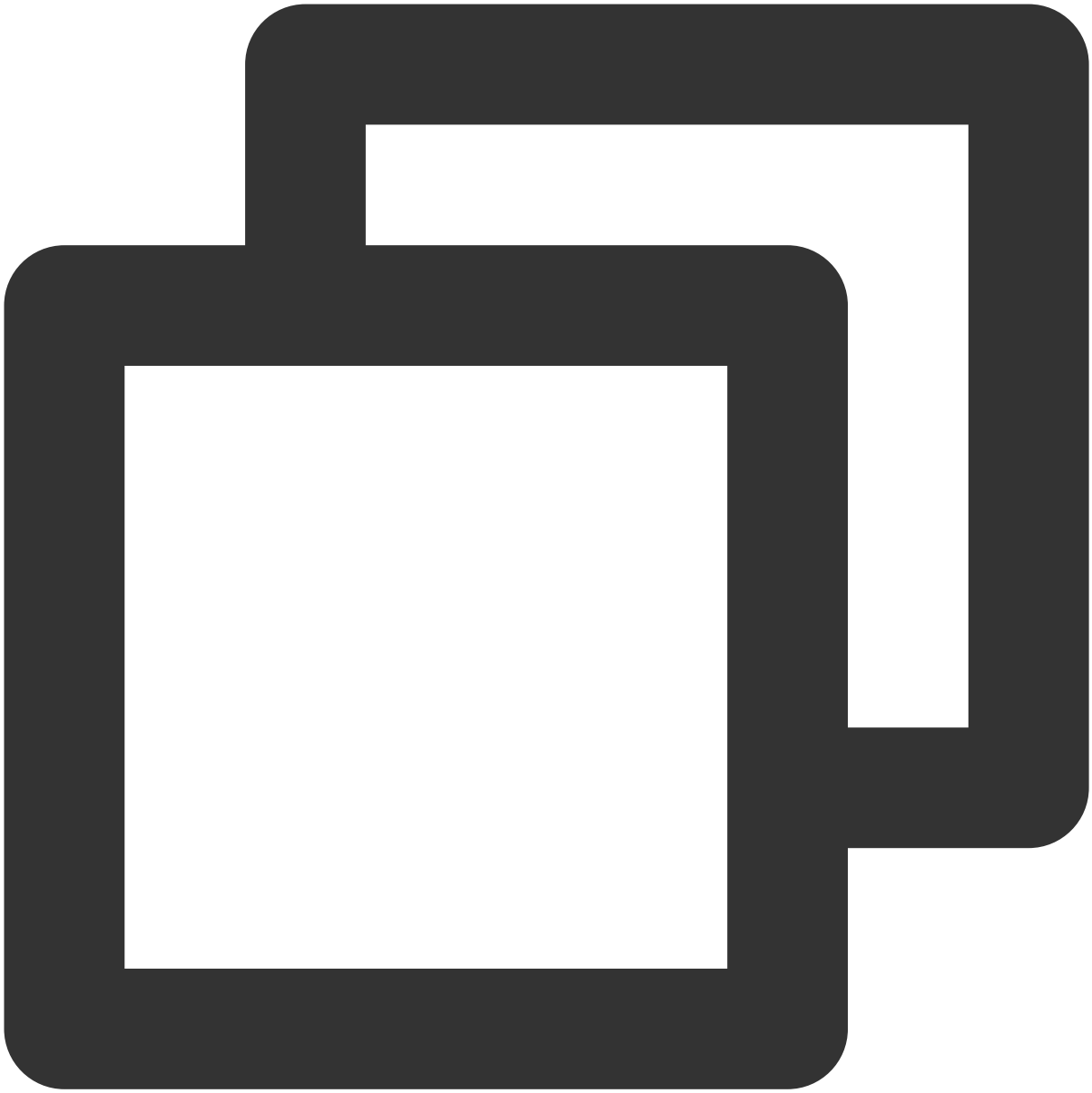
max.poll.records：这个参数可以根据应用的处理能力进行调整。如果您的应用可以处理更多的记录，可以将这个参数设置为更大的值，以减少 poll操作的次数。
auto.commit.interval.ms：这个参数可以根据您的应用的需求进行调整，一般自动提交位点场景，建议保持默认值5000ms。注意，过于频繁的位点提交会影响性能，额外占用 Broker 的计算资源。
client.id：可以为每个消费者设置一个唯一的 ID，以便在监控和日志中区分不同的消费者。
以上是一些常见的参数调优方案，但具体的最佳设置可能会根据您的应用的特性和需求有所不同。在调优参数时，请记住始终进行性能测试，以确保您的设置可以达到预期的效果。
2. 对于 rebalance 时间频繁和消费线程阻塞问题，参考以下说明参数优化：
session.timeout.ms：v0.10.2之前的版本可适当提高该参数值，需要大于消费一批数据的时间，但不要超过30s，建议设置为25s；而v0.10.2及其之后的版本，保持默认值10s即可。
max.poll.records：降低该参数值，建议远远小于<单个线程每秒消费的条数> * <消费线程的个数> * <max.poll.interval.ms>的积。
max.poll.interval.ms：该值要大于<max.poll.records> / (<单个线程每秒消费的条数> * <消费线程的个数>)的值。

创建消费者实例

Kafka Java Client 提供订阅的模型创建消费者，其中在提交位点方面，提供手动提交位点和自动提交位点两种方式。

自动提交位点

自动提交位点：消费者在拉取消息后会自动提交位点，无需手动操作。这种方式的优点是简单易用，但是可能会导致消息重复消费或丢失。注意，自动提交位点时间间隔 auto.commit.interval.ms 不要设置太短，否则容易导致 Broker CPU 偏高，影响其他请求处理。

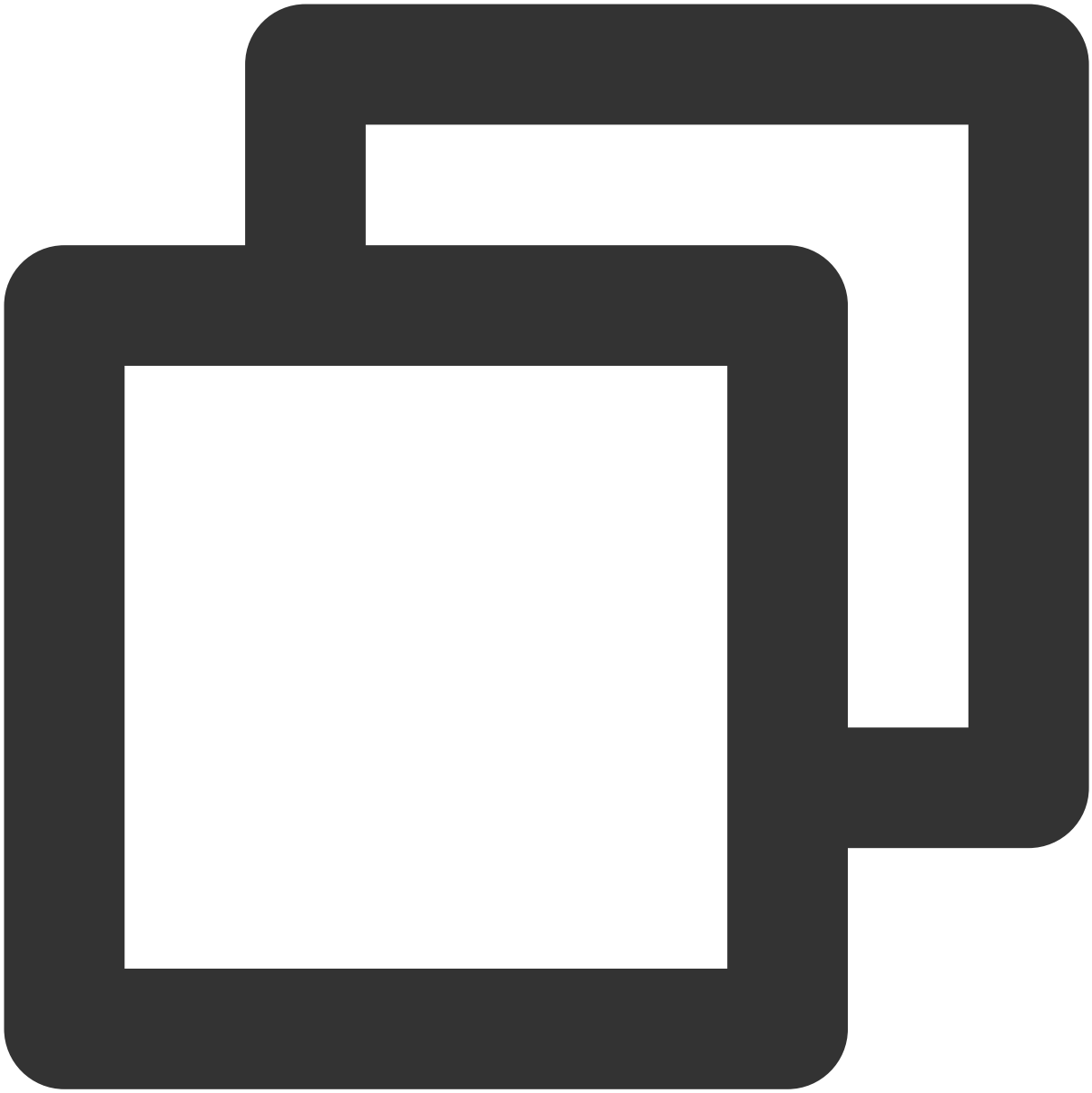


```
import org.apache.kafka.clients.consumer.*;
import org.apache.kafka.common.serialization.StringDeserializer;

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

public class KafkaConsumerAutoCommitExample {

    private static final String BOOTSTRAP_SERVERS = "localhost:9092";
```

```
    private static final String TOPIC = "test-topic";
    private static final String GROUP_ID = "test-group";

    public static void main(String[] args) {
        // 创建Kafka消费者配置
        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS);
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserialize
        props.put(ConsumerConfig.GROUP_ID_CONFIG, GROUP_ID);
        props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");

        // 开启自动提交位点
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, true);
        props.put(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CONFIG, 5000); // 自动提交间

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

        // 订阅主题
        consumer.subscribe(Collections.singletonList(TOPIC));

        // 消费消息
        try {
            while (true) {
                ConsumerRecords<String, String> records = consumer.poll(Duration.of
                for (ConsumerRecord<String, String> record : records) {
                    System.out.printf("消费消息：topic=%s, partition=%d, offset=%d, k
                            record.topic(), record.partition(), record.offset(), re
                }
            }
        } finally {
            // 关闭消费者
            consumer.close();
        }
    }
}
```

手动提交位点

手动提交位点：消费者在处理完消息后需要手动提交位点。这种方式的优点是可以精确控制位点的提交，避免消息重复消费或丢失。但是需要注意，手动提交位点如果太频繁会导致 Broker CPU 很高，影响性能，随着消息量增加，CPU 消费会很高，影响正常 Broker 的其他功能，因此建议间隔一定消息提交位点。

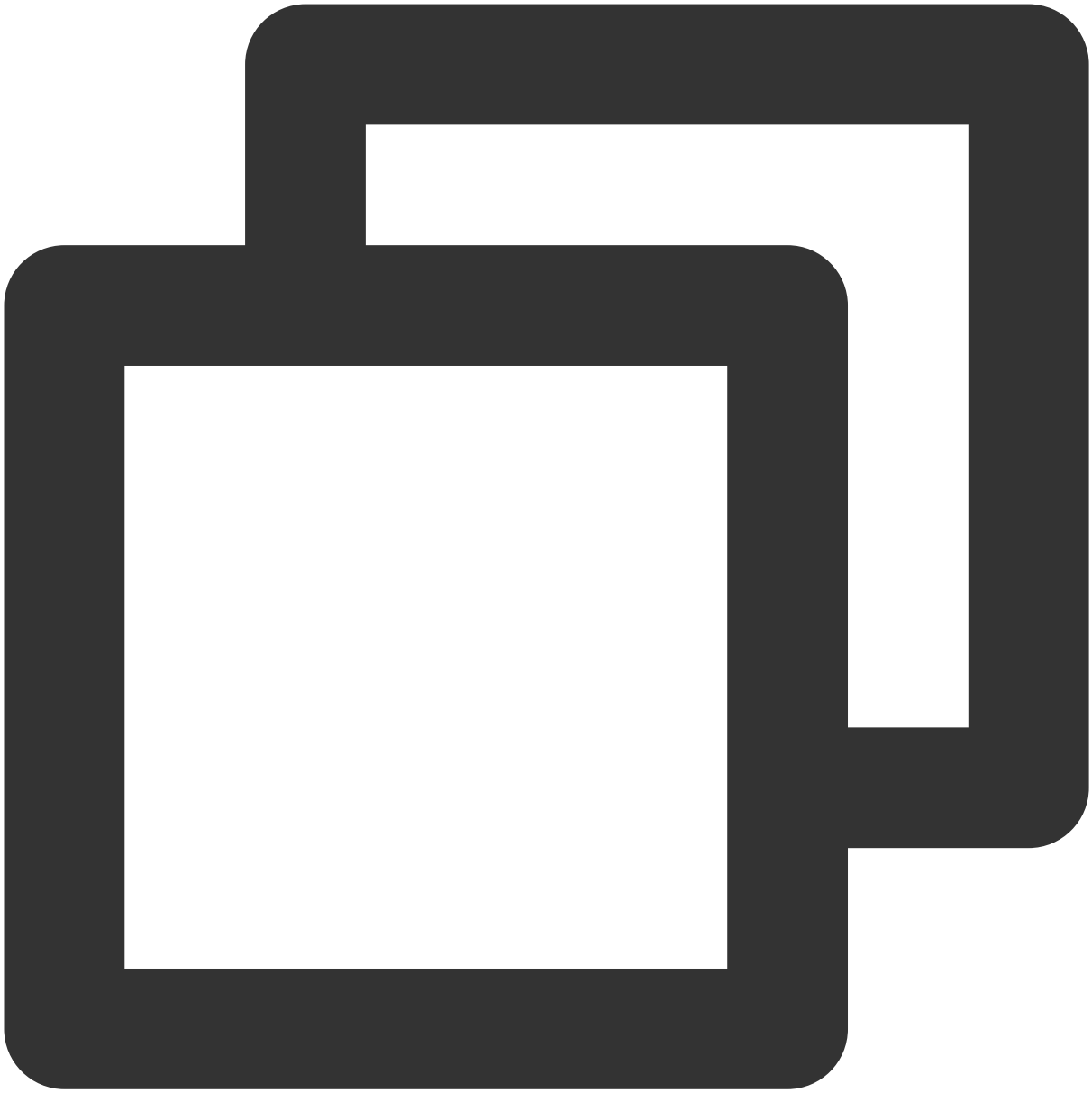


```
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.serialization.StringDeserializer;

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

```
public class KafkaConsumerManualCommitExample {

    public static void main(String[] args) {
        Properties properties = new Properties();
        properties.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        properties.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserial
        properties.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeseri
        properties.put(ConsumerConfig.GROUP_ID_CONFIG, "test-group");
        properties.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
        properties.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "false");

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

        int count = 0;
        try {
            while (true) {
                ConsumerRecords<String, String> records = consumer.poll(Duration.of
                for (ConsumerRecord<String, String> record : records) {
                    System.out.printf("offset = %d, key = %s, value = %s%n", record
                    count++;
                    if (count % 10 == 0) {
                        consumer.commitSync();
                        System.out.println("Committed offsets.");
                    }
                }
            }
        } finally {
            consumer.close();
        }
    }
}
```

Assign 消费

Kafka Java Client 的 assign 消费模式允许消费者直接指定订阅的分区，而不是通过订阅主题来自动分配分区。这种模式适用于需要手动控制消费分区的场景，例如：为了实现特定的负载均衡策略，或者在某些情况下跳过某些分区。一般流程为使用 assign 方法来手动指定消费者消费的分区，通过seek 方法来设置开始消费的位点，然后执行消费逻辑，使用示例如下：



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

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

public class KafkaConsumerAssignAndSeekApp {
```

```
    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.String
        props.put("value.deserializer", "org.apache.kafka.common.serialization.Stri
        props.put("enable.auto.commit", "false");

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

        String topic = "my-topic";
        TopicPartition partition0 = new TopicPartition(topic, 0);
        TopicPartition partition1 = new TopicPartition(topic, 1);
        consumer.assign(Arrays.asList(partition0, partition1));

        // 设置消费的起始位点
        long startPosition0 = 10L;
        long startPosition1 = 20L;
        consumer.seek(partition0, startPosition0);
        consumer.seek(partition1, startPosition1);

        try {
            while (true) {
                ConsumerRecords<String, String> records = consumer.poll(Duration.of
                for (ConsumerRecord<String, String> record : records) {
                    System.out.printf("offset = %d, key = %s, value = %s%n", record
                }
                consumer.commitSync(); // 手动提交位点
            }
        } finally {
            consumer.close();
        }
    }
}
```

Kafka Java Client 生产消费常见问题

Kafka Java Producer 无法成功发送消息

首先排查 Kafka 集群的 IP 和端口能够正常连接，若不能请先解决通信问题。

其次检查是否正确配置接入点，版本是否和 Broker 版本匹配，可以参考最佳实践的发送 demo。

Kafka Python SDK

最近更新时间：2024-07-04 16:00:59

背景

CKafka 的 Python 客户端有以下几个主要的库：
kafka-python：这是一个纯 Python 实现的 Kafka 客户端，支持 Kafka 0.8.2及更高版本。它提供了生产者、消费者和管理 Kafka 集群的 API。这个库易于使用，但性能可能不如基于 librdkafka 的客户端。
安装方法：`pip install kafka-python`
confluent-kafka-python：这个库是基于高性能的 C 库 librdkafka 实现的。它支持Kafka 0.9及更高版本，并提供了生产者、消费者和管理 Kafka 集群的 API。这个库性能更好，但可能需要安装额外的依赖。
安装方法：`pip install confluent-kafka`
aiokafka：这是一个基于 kafka-python 的异步 Kafka 客户端，使用 asyncio 库。这个库适用于需要异步编程的场景。
安装方法：`pip install aiokafka`
pykafka：这是一个支持 Kafka 0.8.x 版本的 Python 客户端。它提供了生产者、消费者和管理 Kafka 集群的 API。这个库已经不再积极维护，但仍然适用于需要支持较旧版本的 Kafka 的场景。
安装方法：`pip install pykafka`
在选择 Python Kafka 客户端时，请根据您的应用需求和 Kafka 版本选择合适的库。对于大多数场景，推荐使用 kafka-python 或 confluent-kafka-python，因为它们支持较新的 Kafka 版本并且功能更完善。如果您的应用需要异步编程，可以考虑使用 aiokafka。
本文重点介绍 kafka-python 的使用方式，官网文档参见 kafka-python。

生产者实践

版本选择

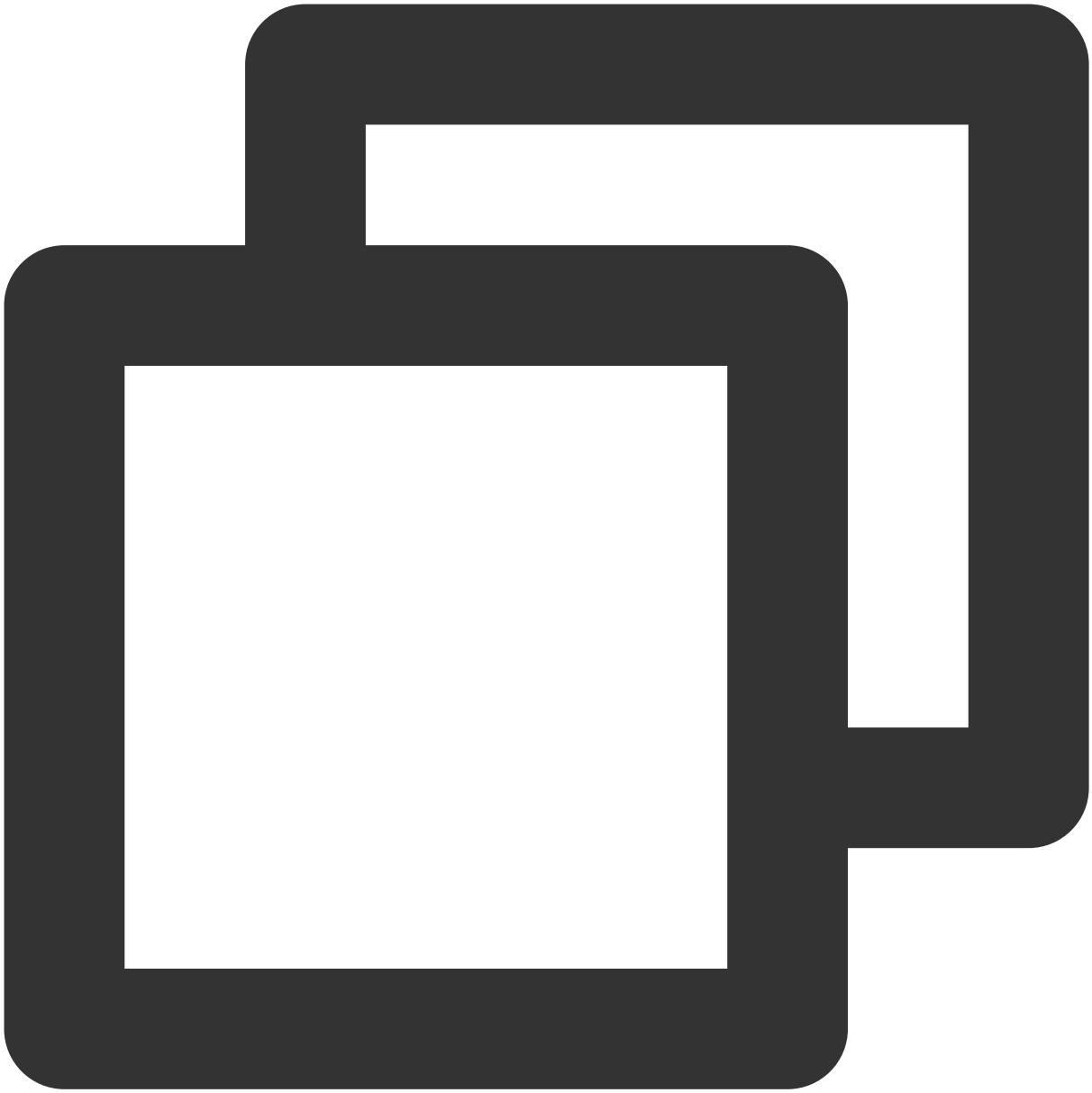
在使用 kafka-python 时，需要先安装 kafka-python 库。可以使用以下命令进行安装：

```
pip install kafka-python
```

生产者参数与调优

生产者参数

Kafka Python 涉及如下关键参数，相关的参数和默认值如下：



```
from kafka import KafkaProducer

producer = KafkaProducer(
    bootstrap_servers='localhost:9092', # 用于初始化连接到Kafka集群的broker列表，默认值为
    client_id=None, # 自定义客户端ID，用于在Kafka服务端日志中识别客户端，默认值为None
    key_serializer=None, # 用于将消息键序列化为字节的可调用对象，默认值为None
    value_serializer=None, # 用于将消息值序列化为字节的可调用对象，默认值为None
    compression_type=None, # 消息压缩类型，可选值为'gzip', 'snappy', 'lz4'或None，表示不
    retries=0, # 重新发送失败的消息的次数，默认值为0
    batch_size=16384, # 用于批处理消息的大小，单位为字节，默认值为16384
    linger_ms=0, # 在批处理消息之前等待更多消息的最长时间，单位为毫秒，默认值为0
```

```
    partitioner=None, # 用于确定消息分区的可调用对象，默认值为None
    buffer_memory=33554432, # 用于缓冲待发送消息的内存总量，单位为字节，默认值为33554432
    connections_max_idle_ms=540000, # 空闲连接的最长保持时间，单位为毫秒，默认值为540000
    max_block_ms=60000, # 在达到缓冲区内存限制时，send()方法阻塞的最长时间，单位为毫秒，默认值
    max_request_size=1048576, # 发送到broker的请求的最大字节数，默认值为1048576
    metadata_max_age_ms=300000, # 元数据在本地缓存中的最长存活时间，单位为毫秒，默认值为3000
    retry_backoff_ms=100, # 两次重试之间的等待时间，单位为毫秒，默认值为100
    request_timeout_ms=30000, # 客户端等待请求响应的最长时间，单位为毫秒，默认值为30000
    receive_buffer_bytes=32768, # 用于接收数据的网络缓冲区大小，单位为字节，默认值为32768
    send_buffer_bytes=131072, # 用于发送数据的网络缓冲区大小，单位为字节，默认值为131072
    acks='all', # 消息确认机制，可选值为'0', '1', 或'all'，默认值为'all'
    transactional_id=None, # 事务ID，用于标识生产者参与事务的唯一标识，默认值为None
    transaction_timeout_ms=60000, # 事务超时时间，单位为毫秒，默认值为60000
    enable_idempotence=False, # 是否启用幂等性，默认值为False
    security_protocol='PLAINTEXT', # 安全协议类型，可选值为'PLAINTEXT', 'SSL', 'SASL_PI
```

参数说明调优

关于 acks 参数优化

acks 参数用于控制生产者发送消息时的确认机制。该参数的默认值为-1，表示消息发送给 Leader Broker 后，Leader 确认以及相应的 Follower 消息都写入完成后才返回。acks 参数还有以下可选值：0，1，-1。在跨可用区场景，以及副本数较多的 Topic，acks 参数的取值会影响消息的可靠性和吞吐量。因此：

在一些在线业务消息的场景下，吞吐量要求不大，可以将 acks参 数设置为-1，则可以确保消息被所有副本接收和确认后才返回，从而提高消息的可靠性。

在日志采集等大数据或者离线计算的场景下，要求高吞吐（即每秒写入 Kafka 的数据量）的情况下，可以将 acks 设置为1，提高吞吐量。

关于 buffer_memory 参数优化（缓存）

默认情况下，传输同等数据量的情况下，多次请求和一次请求的网络传输，一次请求传输能有效减少相关计算和网络资源，提高整体写入的吞吐量。

因此，可以通过这个参数设置优化客户端发送消息的吞吐能力。对于 Kafka Python Client，默认提供 linger_ms 为 0ms 的攒批时间积攒消息，此处可以优化，适当增加值，例如设置100ms，进行聚合多个请求批量发送消息，提高吞吐。如果带宽较高，且单机内存充足，建议调大 buffer_memory 提高吞吐。

关于压缩参数优化

Kafka Python Client 支持如下压缩参数：none, gzip, snappy, lz4。

none：不使用压缩。

gzip：使用 GZIP 压缩。

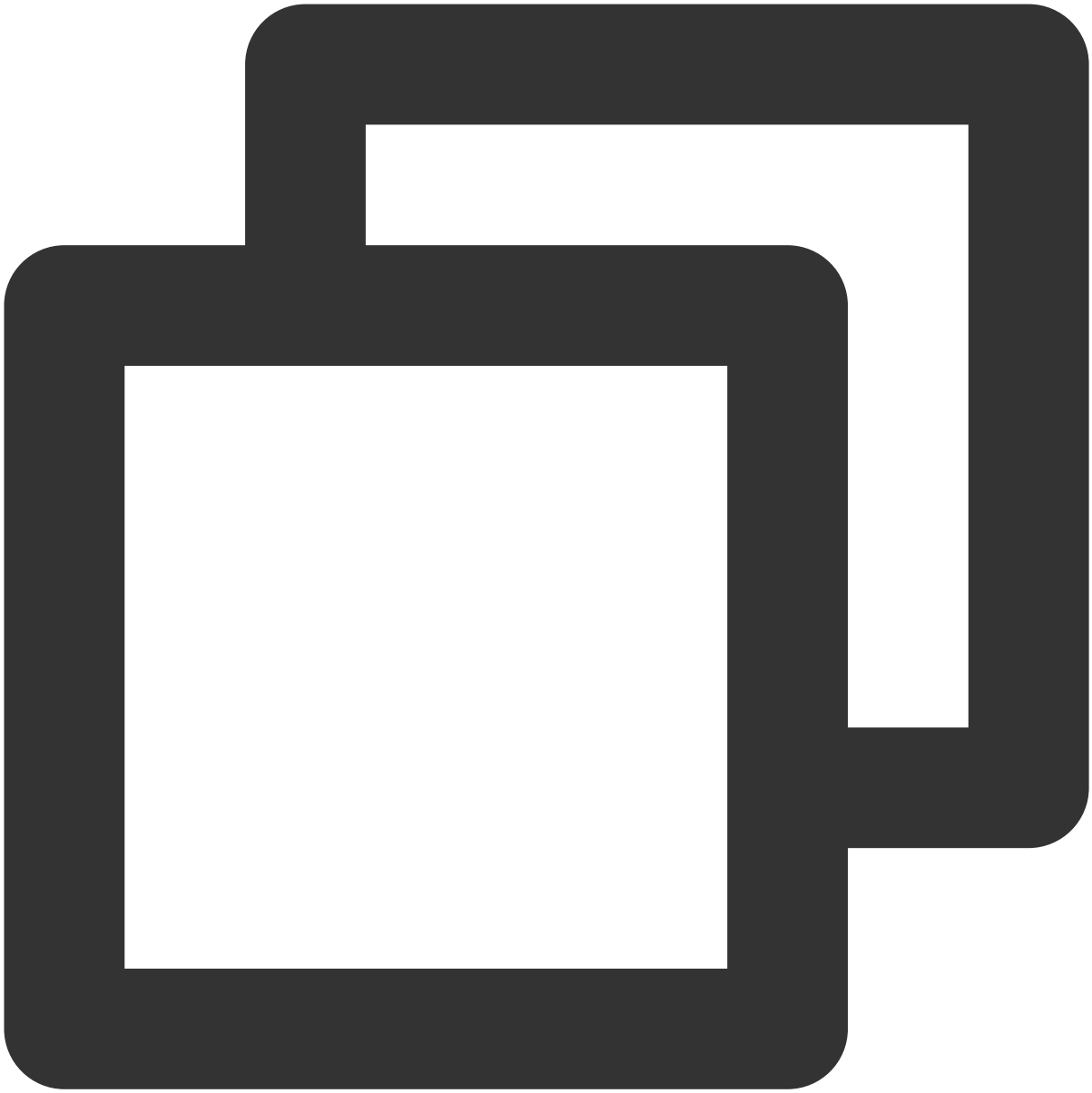
snappy：使用 Snappy 压缩。

lz4：使用 LZ4 压缩。

要在 Producer 客户端中使用压缩消息，需要在创建生产者时设置 compression_type 参数。例如，要使用 LZ4 压缩算法，可以将 compression_type 设置为 lz4，虽然压缩消息的压缩和解压缩，发生客户端，是一种用计算换带宽的优化方式，但是由于 Broker 针对压缩消息存在校验行为会付出额外的计算成本，在低流量情况，不建议使用压缩，尤其是 gzip 压缩，Broker 校验计算成本会比较大，在某种程度上可能会出现得不偿失的情况，反而因为计算的增加导致 Broker 处理其他请求的能力偏低，导致带宽吞吐更低。这种情况建议可以使用如下方式：
在 Producer 端对消息数据独立压缩，生成压缩包数据：messageCompression，同时在消息的 key 存储压缩方式：`{"Compression","lz4"}`
在 Producer 端将 messageCompression 当成正常消息发送。
在 Consumer 端读取消息 key，获取使用的压缩方式，独立进行解压缩。

创建生产者实例

如果应用程序需要更高的可靠性，则可以使用同步生产者，以确保消息发送成功。同时，可以使用 ACK 确认机制和事务机制，以确保消息的可靠性和一致性。具体的参数调优参考生产者参数与调优。如果应用程序需要更高的吞吐量，则可以使用异步生产者，以提高消息的发送速度。同时，可以使用批量发送消息的方式，以减少网络开销和 IO 消耗。示例如下：



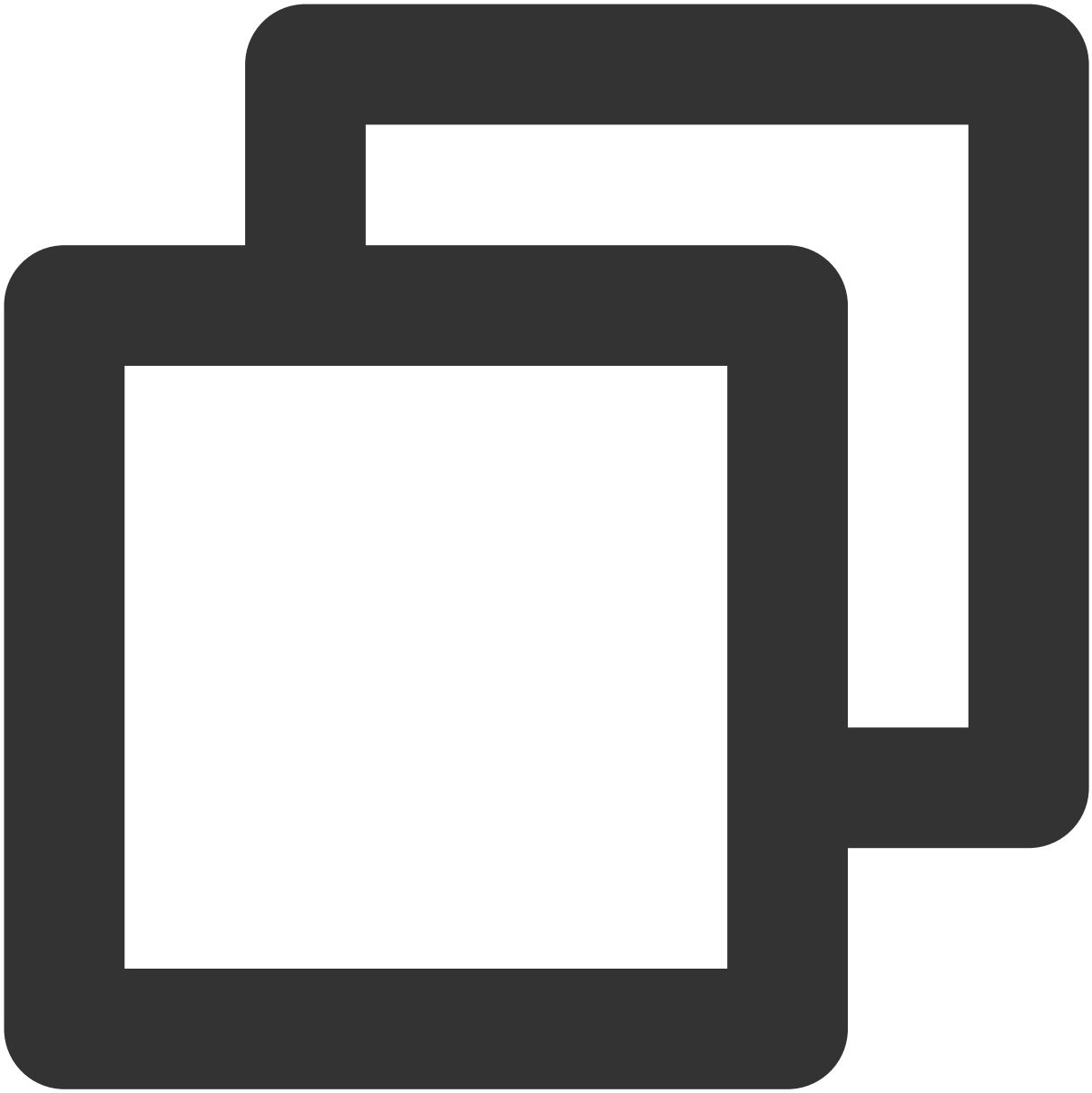
```
from kafka import KafkaProducer
import sys

# 参数配置
BOOTSTRAP_SERVERS = 'localhost:9092'
TOPIC = 'test_topic'
SYNC = True
ACKS = '1'  # leader副本确认写入即可
LINGER_MS = 500  # 延迟500ms发送
BATCH_SIZE = 16384  # 消息批次大小16KB
```

```
def create_producer(servers, acks, linger_ms, batch_size):
    return KafkaProducer(bootstrap_servers=servers, acks=acks, linger_ms=linger_ms,

def send_message_sync(producer, topic, message):
    future = producer.send(topic, message)
    result = future.get(timeout=10)
    print(f"Sent message: {message} to topic: {topic}, partition: {result.partition

def send_message_async(producer, topic, message):
    def on_send_success(record_metadata):
        print(f"Sent message: {message} to topic: {topic}, partition: {record_metad

    def on_send_error(excp):
        print(f"Error sending message: {message} to topic: {topic}", file=sys.stder
        print(excp, file=sys.stderr)

    future = producer.send(topic, message)
    future.add_callback(on_send_success).add_errback(on_send_error)

def main():
    producer = create_producer(BOOTSTRAP_SERVERS, ACKS, LINGER_MS, BATCH_SIZE)
    messages = ['Hello Kafka', 'Async vs Sync', 'Demo']

    if SYNC:
        for message in messages:
            send_message_sync(producer, TOPIC, message.encode('utf-8'))
    else:
        for message in messages:
            send_message_async(producer, TOPIC, message.encode('utf-8'))

    producer.flush()

if __name__ == '__main__':
    main()
```

消费者实践

消费者参数与调优

消费者参数



```
from kafka import KafkaConsumer

# 创建一个KafkaConsumer对象，用于连接Kafka集群并消费消息
consumer = KafkaConsumer(
    'topic_name', # 要订阅的主题列表
    bootstrap_servers=['localhost:9092'],     # Kafka集群的接入点
    group_id=None,     # 消费者组ID，用于将消费者分组，要加入动态分区分配（如果启用）并用于获取
    client_id='kafka-python-{version}',#客户端Id,默认是kafka-python-{version}
    api_version=None, #指定要使用的 Kafka API 版本。如果设置为 None，客户端将尝试通过API请求
    enable_auto_commit=True,     # 是否自动提交消费位置，默认为True
    auto_commit_interval_ms=5000,     # 自动提交消费位置的间隔，默认为5秒（5000毫秒）
```

```
    auto_offset_reset='latest',     # 消费者在读取的分区中的消费位置的策略，默认为'latest'（
    fetch_min_bytes=1,     # 消费者在读取分区时的最小字节数，默认为1字节
    fetch_max_wait_ms=500,     # 在没有新的消费数据，默认等待500ms
    fetch_max_bytes=52428800,     # 消费者在读取分区时的最大字节数，默认为52428800字节（50MB
    max_poll_interval_ms=300000     # 消参数默认值为300000毫秒（5分钟）。如果消费者在5分钟内
    retry_backoff_ms=100,     # 重试间隔时间，默认为100毫秒
    reconnect_backoff_max_ms=1000,     # 重新连接到多次连接失败的Broker的间隔时间最大值（以
    request_timeout_ms=305000,     # 客户端请求超时（以毫秒为单位）
    session_timeout_ms=10000,     # session_timeout_ms (int) - 使用 Kafka 组管理工具时用
    heartbeat_interval_ms=3000,     # 使用 Kafka 的组管理工具时，向消费者协调器发出心跳之间的
    receive_buffer_bytes=32768,#读取数据时使用的 TCP 接收缓冲区 (SO_RCVBUF) 的大小。默认值
    send_buffer_bytes=131072# 发送数据时使用的 TCP 发送缓冲区 (SO_SNDBUF) 的大小。默认值：
)

for message in consumer:
    print(f"Topic: {message.topic}, Partition: {message.partition}, Offset: {messag
```

参数说明与调优

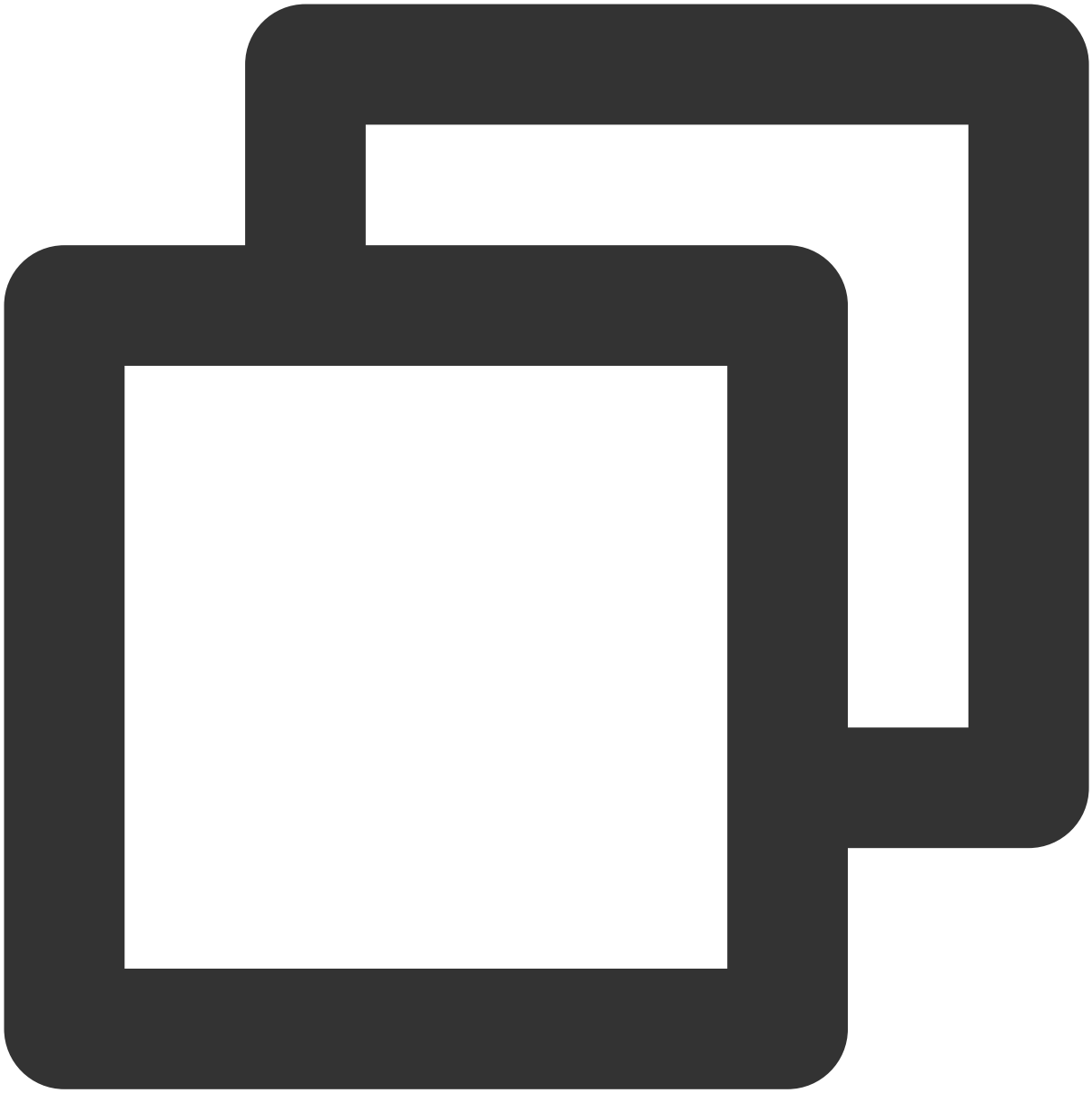
1. max_poll_interval_ms 是 Kafka Python Consumer 的一个配置参数，它用于指定 Consumer 在两次 poll 操作之间的最大延迟。这个参数的主要作用是控制 Consumer 的 存活，也就是判断 Consumer 是否还活着。如果 Consumer 在 max_poll_interval_ms 指定的时间内没有进行 poll 操作，那么 Kafka 认为这个 Consumer 已经挂掉，会触发 Consumer 的 rebalance 操作。这个参数的设置需要根据实际的消费速度来调整。如果设置得太小，可能会导致 Consumer 频繁地触发 rebalance 操作，增加了 Kafka 的负担；如果设置得太大，可能会导致 Consumer 在出现问题时不能及时被 Kafka 检测到，从而影响了消息的消费。建议在高吞吐下可以时长增加该值的设置。
2. 针对自动提交位点请求，建议 auto_commit_interval_ms 时间不要低于1000ms，因为频率过高的位点请求会导致 Broker CPU 很高，影响其他正常服务的读写。

创建消费者实例

Kafka Python 提供订阅的模型创建消费者，其中在提交位点方面，提供手动提交位点和自动提交位点两种方式。

自动提交位点

自动提交位点：消费者在拉取消息后会自动提交位点，无需手动操作。这种方式的优点是简单易用，但是可能会导致消息重复消费或丢失。建议间隔5s提交位点。

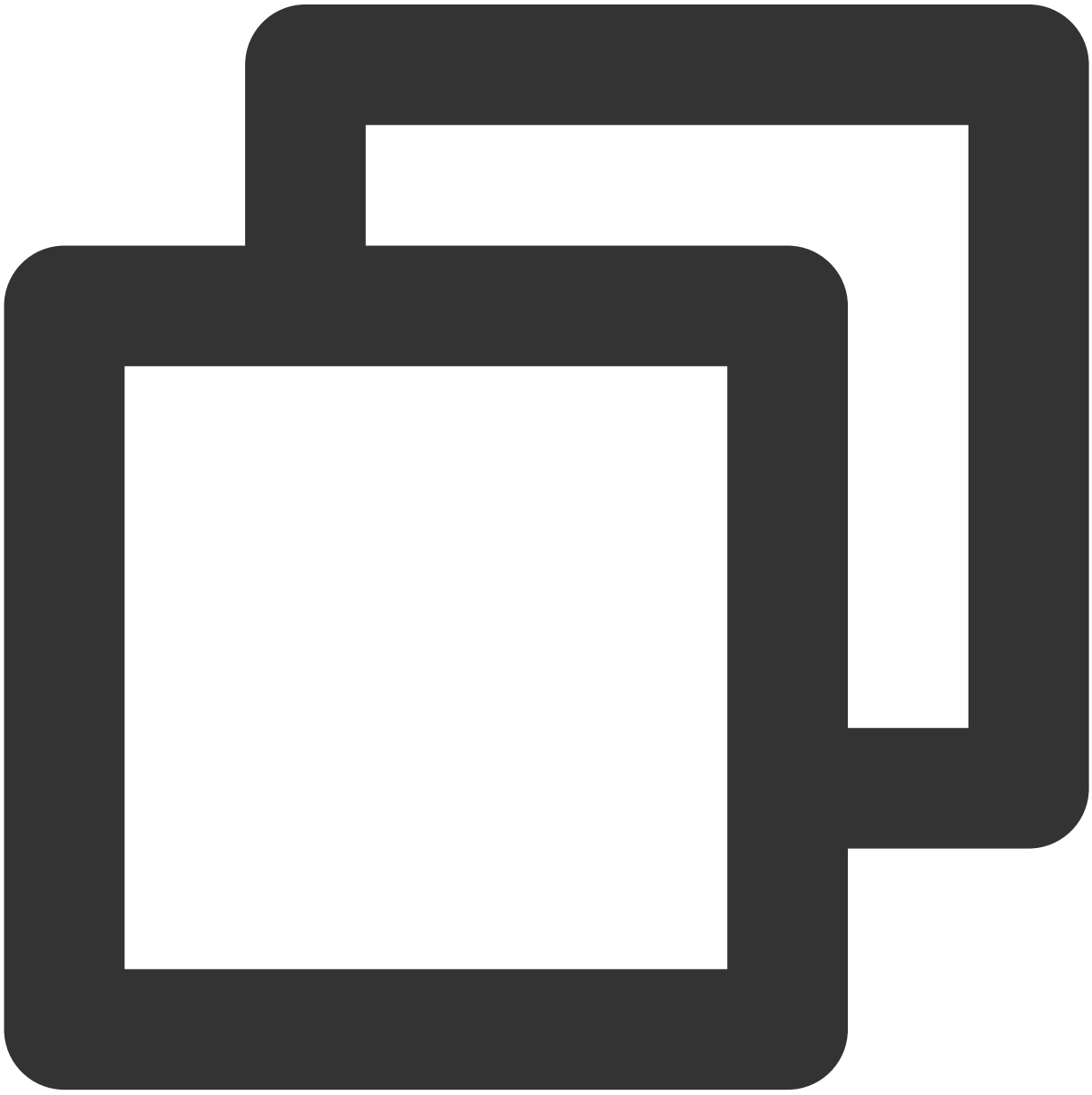


```
# auto_commit_consumer_interval.py
from kafka import KafkaConsumer
from time import sleep

consumer = KafkaConsumer(
    'your_topic_name',
    bootstrap_servers=['localhost:9092'],
    group_id='auto_commit_group',
    auto_commit_interval_ms=5000   # 设置自动提交位点的间隔为5000毫秒（5秒）
)
```

```
for message in consumer:
    print(f"Topic: {message.topic}, Partition: {message.partition}, Offset: {messag
    sleep(1)
```

手动提交位点

手动提交位点：消费者在处理完消息后需要手动提交位点。这种方式的优点是可以精确控制位点的提交，避免消息重复消费或丢失。但是需要注意，手动提交位点如果太频繁会导致 Broker CPU 很高，影响性能，随着消息量增加，CPU 消费会很高，影响正常 Broker 的其他功能，因此建议间隔一定消息提交位点。



```
# manual_commit_consumer.py
from kafka import KafkaConsumer
from kafka.errors import KafkaError
from time import sleep

consumer = KafkaConsumer(
    'your_topic_name',
    bootstrap_servers=['localhost:9092'],
    group_id='manual_commit_group',
    enable_auto_commit=False
)

count = 0
for message in consumer:
    print(f"Topic: {message.topic}, Partition: {message.partition}, Offset: {messag
    count += 1

    if count % 10 == 0:
        try:
            consumer.commit()
        except KafkaError as e:
            print(f"Error while committing offset: {e}")

    sleep(1)
```

librdkafka SDK

最近更新时间：2024-07-04 16:00:59

背景

TDMQ CKafka 是一个分布式流处理平台，用于构建实时数据管道和流式应用程序。它提供了高吞吐量、低延迟、可伸缩性和容错性等特性。
本文着重介绍上述 librdkafka 客户端的关键参数和实践教程，以及常见问题。

生产者实践

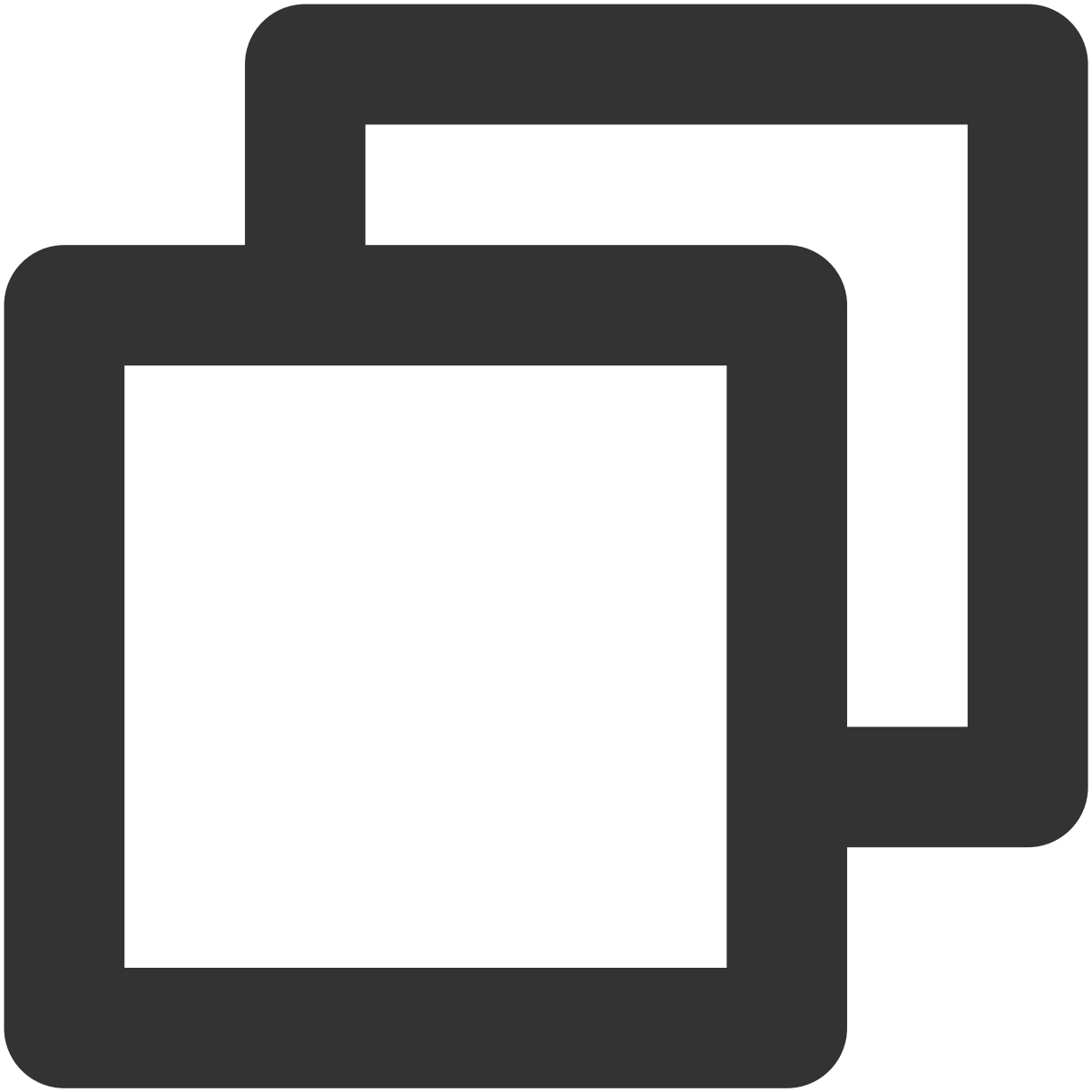
版本选择

在使用 librdkafka 时，librdkafka 会自动根据 Kafka 集群的版本选择适当的协议版本进行通信，由于 kafka 的版本迭代更新较快，通常情况下，使用最新的 librdkafka 版本可以获得更好的兼容性和性能。

生产者参数与调优

生产者参数

librdkafka 主要涉及如下关键参数，相关的参数和默认值如下：



```
rd_kafka_conf_t *conf = rd_kafka_conf_new();

// Kafka集群的地址，多个地址用逗号分隔，默认为空
rd_kafka_conf_set(conf, "bootstrap.servers", "localhost:9092", NULL, 0);

// 发送消息的最大尝试次数，包括第一次尝试，默认为2
rd_kafka_conf_set(conf, "message.send.max.retries", "2", NULL, 0);

// 重试之间的回退时间（以毫秒为单位），默认为100
rd_kafka_conf_set(conf, "retry.backoff.ms", "100", NULL, 0);
```

```
// 客户端请求超时时间（以毫秒为单位），默认为5000
rd_kafka_conf_set(conf, "request.timeout.ms", "5000", NULL, 0);

// 客户端发送缓冲区大小（以字节为单位），默认为131072
rd_kafka_conf_set(conf, "queue.buffering.max.kbytes", "131072", NULL, 0);

// 客户端发送缓冲区中消息的最大数量，默认为100000
rd_kafka_conf_set(conf, "queue.buffering.max.messages", "100000", NULL, 0);

// 客户端发送缓冲区中消息的最大总大小（以字节为单位），默认为1000000
rd_kafka_conf_set(conf, "queue.buffering.max.total.bytes", "1000000", NULL, 0);

// 客户端发送缓冲区的linger时间（以毫秒为单位），默认为0
rd_kafka_conf_set(conf, "queue.buffering.max.ms", "0", NULL, 0);

// 是否启用消息压缩，默认为0（不启用）
rd_kafka_conf_set(conf, "compression.codec", "none", NULL, 0);

// 消息压缩级别，默认为0（自动选择）
rd_kafka_conf_set(conf, "compression.level", "0", NULL, 0);

// 客户端的ID，默认为rdkafka
rd_kafka_conf_set(conf, "client.id", "rdkafka", NULL, 0);

// 生产者的最大并发请求数，即未收到broker响应的请求数，默认为1000000
rd_kafka_conf_set(conf, "max.in.flight.requests.per.connection", "1000000", NULL, 0

// 客户端与Kafka集群的连接最大重试次数，默认为3次
rd_kafka_conf_set(conf, "broker.address.ttl", "3", NULL, 0);

// 客户端与Kafka集群的连接重试间隔（以毫秒为单位），默认为1000
rd_kafka_conf_set(conf, "reconnect.backoff.ms", "1000", NULL, 0);

// 客户端与Kafka集群的连接重试最大间隔（以毫秒为单位），默认为10000
rd_kafka_conf_set(conf, "reconnect.backoff.max.ms", "10000", NULL, 0);

// 客户端API版本的回退时间（以毫秒为单位），默认为10000
rd_kafka_conf_set(conf, "api.version.request.timeout.ms", "10000", NULL, 0);

// 安全协议，默认为plaintext
rd_kafka_conf_set(conf, "security.protocol", "plaintext", NULL, 0);

// 其他SSL和SASL相关参数，请参考librdkafka官方文档

// 创建生产者实例
rd_kafka_t *producer = rd_kafka_new(RD_KAFKA_PRODUCER, conf, NULL, 0);
```

参数说明调优

关于 acks 参数优化

acks 参数用于控制生产者发送消息时的确认机制。该参数的默认值为-1，表示消息发送给 Leader Broker 后，Leader 确认以及相应的 Follower 消息都写入完成后才返回。acks 参数还有以下可选值：0，1，-1。在跨可用区场景，以及副本数较多的 Topic，acks 参数的取值会影响消息的可靠性和吞吐量。因此：
在一些在线业务消息的场景下，吞吐量要求不大，可以将 acks 参数设置为-1，则可以确保消息被所有副本接收和确认后才返回，从而提高消息的可靠性。
在日志采集等大数据或者离线计算的场景下，要求高吞吐（即每秒写入 Kafka 的数据量）的情况下，可以将 acks 设置为1，提高吞吐。

关于 buffering 参数优化（缓存）

默认情况下，传输同等数据量的情况下，多次请求和一次请求的网络传输，一次请求传输能有效减少相关计算和网络资源，提高整体写入的吞吐量。
因此，可以通过这个参数设置优化客户端发送消息的吞吐能力。对 librdkafka，默认提供5ms的攒批时间积攒消息。如果消息较小，可以适当增加queue.buffering.max.ms 的时间。

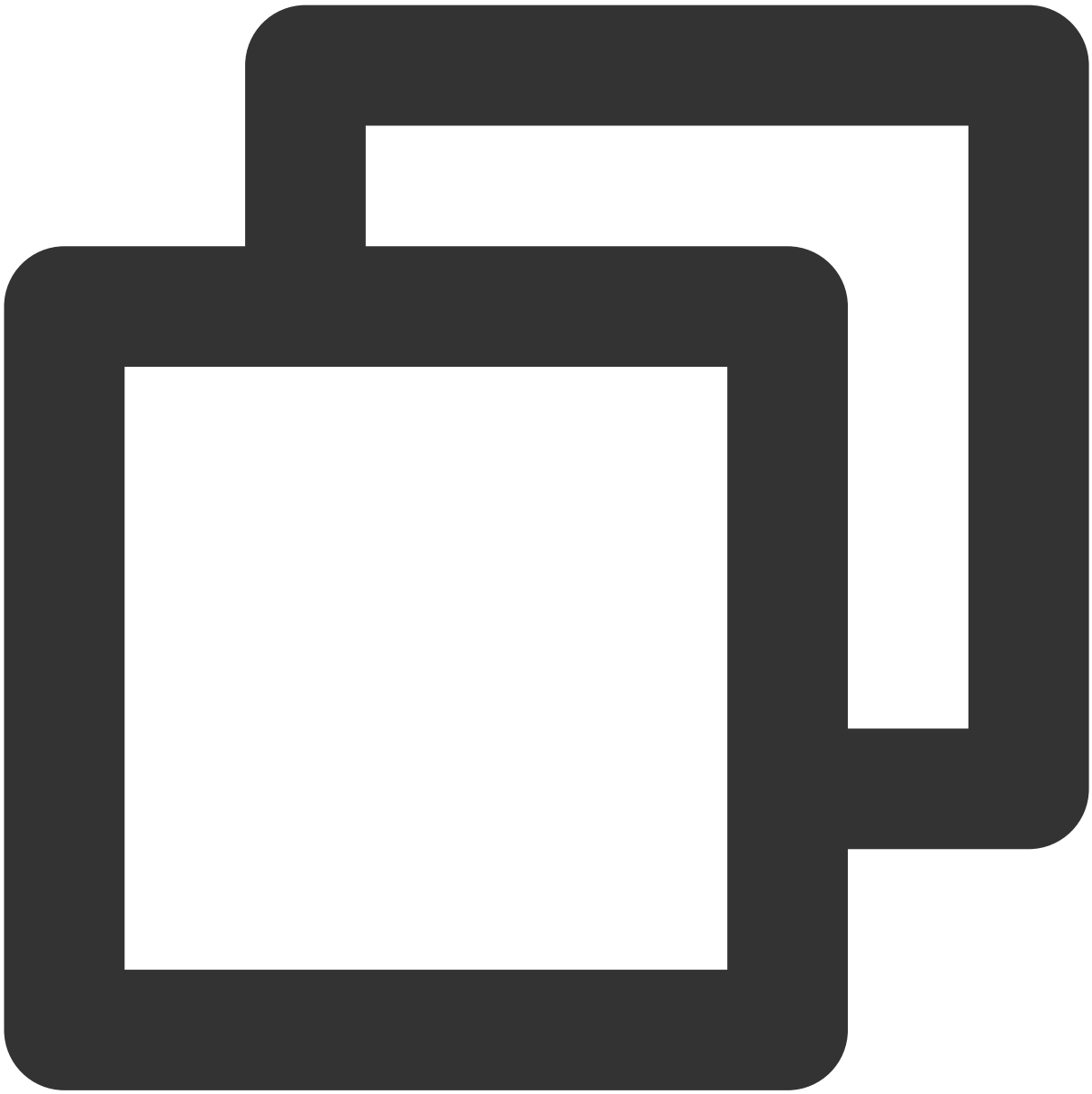
关于压缩参数优化

librdkafka 支持如下压缩参数：none, gzip, snappy, lz4, zstd。
在 librdkafka客户端中，支持以下几种压缩算法：
none：不使用压缩算法。
gzip：使用 GZIP 压缩算法。
snappy：使用 Snappy 压缩算法。
lz4：使用 LZ4 压缩算法。
zstd：使用 ZSTD 压缩算法。
要在 Producer 客户端中使用压缩算法，需要在创建生产者时设置 compression.type 参数。例如，要使用 LZ4 压缩算法，可以将 compression.type 设置为 lz4，虽然压缩算法的 CPU 压缩和 CPU解压缩，发生客户端，是一种用计算换带宽的优化方式，但是由于 Broker 针对压缩消息存在校验行为会付出额外的计算成本，尤其是 Gzip 压缩，服务端的压缩计算成本会比较大，在某种程度上可能会出现得不偿失的情况，反而因为计算的增加导致 Broker 消息处理能力偏低，导致带宽吞吐更低。这种情况建议可以使用如下方式进行使用：
在 Producer 端对消息数据独立压缩，生成压缩包数据：messageCompression，同时在消息的 key 存储压缩方式：`{"Compression","CompressionLZ4"}`
在 Producer 端将 messageCompression 当成正常消息发送。
在 Consumer 端读取消息 key，获取使用的压缩方式，独立进行解压缩。

创建生产者实例

如果应用程序需要更高的吞吐量，则可以使用异步生产者，以提高消息的发送速度。同时，可以使用批量发送消息的方式，以减少网络开销和 IO 消耗。如果应用程序需要更高的可靠性，则可以使用同步生产者，以确保消息发送成

功。同时，可以使用 ACK 确认机制和事务机制，以确保消息的可靠性和一致性。具体的参数调优参考生产者参数与调优。



```
#include <stdio.h>
#include <string.h>
#include <librdkafka/rdkafka.h>

// 生产者消息发送回调
void dr_msg_cb(rd_kafka_t *rk, const rd_kafka_message_t *rkmessage, void *opaque) {
    if (rkmessage->err) {
        fprintf(stderr, "Message delivery failed: %s\\n", rd_kafka_err2str(rkmessag
```

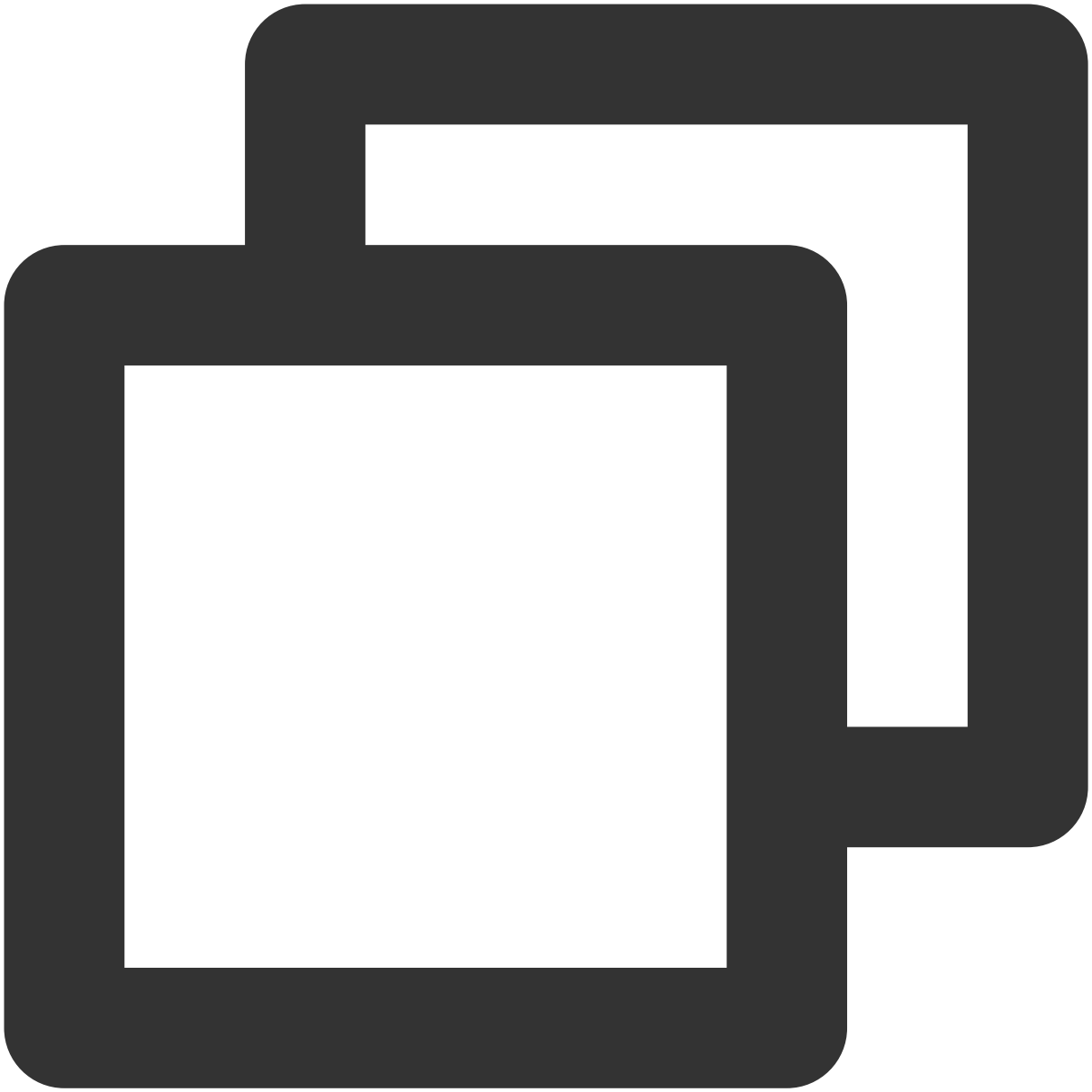
```
    } else {
        fprintf(stderr, "Message delivered (%zd bytes, partition %"PRId32")\\n",
                rkmessage->len, rkmessage->partition);
    }
}

int main() {
    rd_kafka_conf_t *conf = rd_kafka_conf_new();

    // 设置Kafka集群的地址
    rd_kafka_conf_set(conf, "bootstrap.servers", "localhost:9092", NULL, 0);

    // 设置ack等于1，表示leader副本收到消息后即认为发送成功
    rd_kafka_conf_set(conf, "acks", "1", NULL, 0);

    // 设置生产者消息发送回调
    rd_kafka_conf_set_dr_msg_cb(conf, dr_msg_cb);

    // 创建生产者实例
    char errstr[512];
    rd_kafka_t *producer = rd_kafka_new(RD_KAFKA_PRODUCER, conf, errstr, sizeof(err
    if (!producer) {
        fprintf(stderr, "Failed to create producer: %s\\n", errstr);
        return 1;
    }

    // 创建主题实例
    rd_kafka_topic_t *topic = rd_kafka_topic_new(producer, "test", NULL);
    if (!topic) {
        fprintf(stderr, "Failed to create topic: %s\\n", rd_kafka_err2str(rd_kafka_
        rd_kafka_destroy(producer);
        return 1;
    }

    // 发送消息
    const char *message = "Hello, Kafka!";
    if (rd_kafka_produce(
            topic,
            RD_KAFKA_PARTITION_UA,
            RD_KAFKA_MSG_F_COPY,
            (void *)message,
            strlen(message),
            NULL,
            0,
            NULL) == -1) {
        fprintf(stderr, "Failed to produce to topic %s: %s\\n", rd_kafka_topic_name
    }
```

```
    // 等待所有消息发送完成
    while (rd_kafka_outq_len(producer) > 0) {
        rd_kafka_poll(producer, 1000);
    }

    // 销毁主题实例
    rd_kafka_topic_destroy(topic);

    // 销毁生产者实例
    rd_kafka_destroy(producer);

    return 0;
}
```

消费者实践

消费者参数与调优

消费者参数



```
rd_kafka_conf_t *conf = rd_kafka_conf_new();

// 设置Kafka集群的地址
rd_kafka_conf_set(conf, "bootstrap.servers", "localhost:9092", NULL, 0);

// 设置消费组ID，默认为空
rd_kafka_conf_set(conf, "group.id", "mygroup", NULL, 0);

// 设置消费者的自动提交间隔（以毫秒为单位），默认为5000
rd_kafka_conf_set(conf, "auto.commit.interval.ms", "5000", NULL, 0);
```

```
// 设置消费者的自动提交开关，默认为true
rd_kafka_conf_set(conf, "enable.auto.commit", "true", NULL, 0);

// 设置消费者的自动偏移量重置策略，默认为latest
rd_kafka_conf_set(conf, "auto.offset.reset", "latest", NULL, 0);

// 设置客户端的ID，默认为rdkafka
rd_kafka_conf_set(conf, "client.id", "rdkafka", NULL, 0);

// 创建消费者实例
char errstr[512];
rd_kafka_t *consumer = rd_kafka_new(RD_KAFKA_CONSUMER, conf, errstr, sizeof(err
```

参数说明与调优

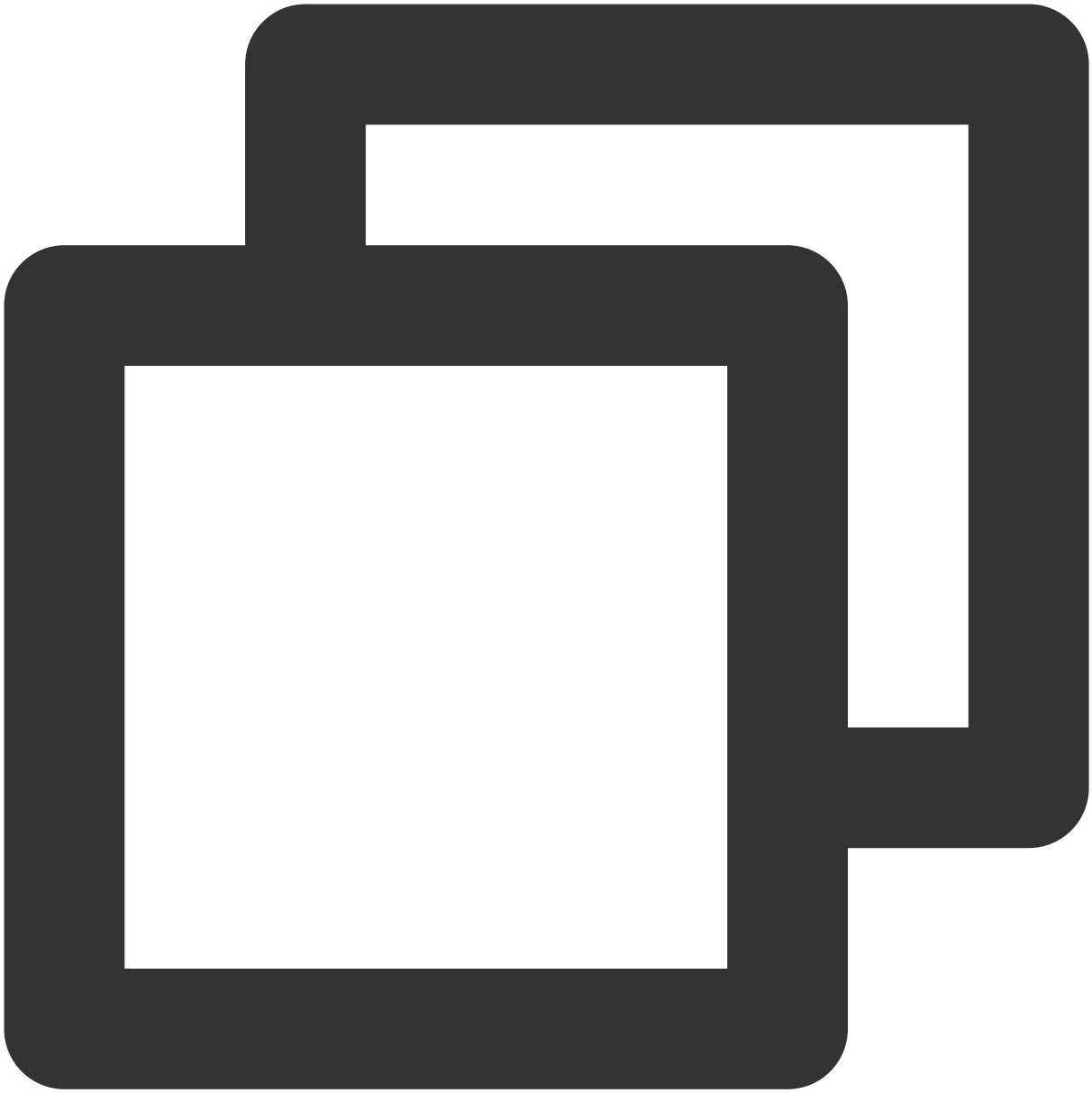
针对自动提交位点请求，建议 auto.commit.interval.ms 时间不要低于1000ms，因为频率过高的位点请求会导致 Broker CPU 很高，影响其他正常服务的读写。

创建消费者实例

提供订阅的模型创建消费者，其中在提交位点方面，提供手动提交位点和自动提交位点两种方式。

自动提交位点

自动提交位点：消费者在拉取消息后会自动提交位点，无需手动操作。这种方式的优点是简单易用，但是可能会导致消息重复消费或丢失。



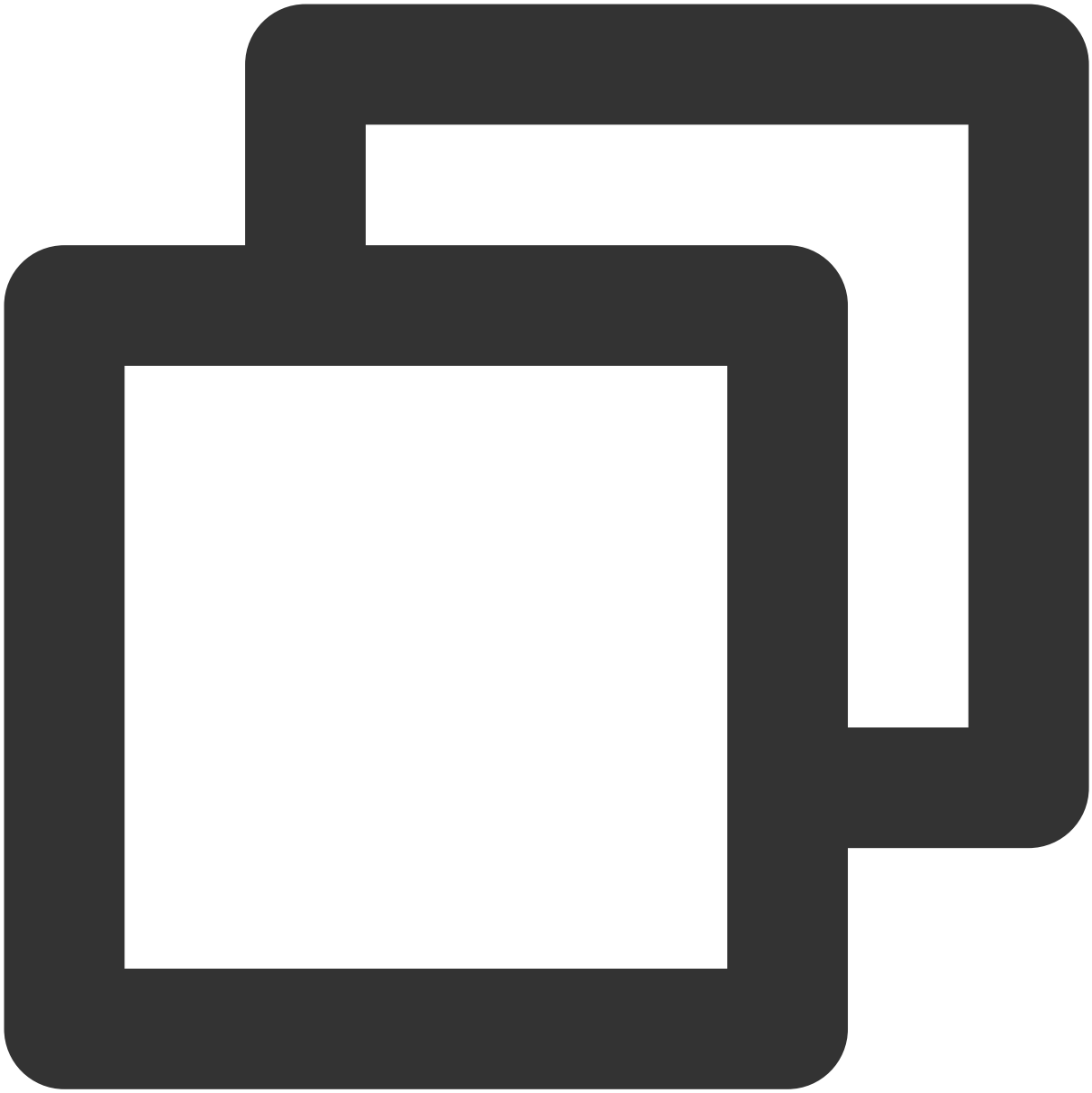
```
#include <stdio.h>
#include <string.h>
#include <librdkafka/rdkafka.h>

// 消费者消息处理回调
void msg_consume(rd_kafka_message_t *rkmessage, void *opaque) {
    if (rkmessage->err) {
        fprintf(stderr, "%% Consume error for topic \\"%s\\" [%"PRId32"] "
                "offset %"PRId64": %s\\n",
                rd_kafka_topic_name(rkmessage->rkt),
                rkmessage->partition, rkmessage->offset,
```

```
                rd_kafka_message_errstr(rkmessage));
    } else {
        printf("%% Message received on topic %s [%"PRId32"] at offset %"PRId64": %.
               rd_kafka_topic_name(rkmessage->rkt),
               rkmessage->partition,
               rkmessage->offset, (int)rkmessage->len, (const char *)rkmessage->pay
    }
}

int main() {
    rd_kafka_conf_t *conf = rd_kafka_conf_new();

    // 设置Kafka集群的地址
    rd_kafka_conf_set(conf, "bootstrap.servers", "localhost:9092", NULL, 0);

    // 设置消费组ID
    rd_kafka_conf_set(conf, "group.id", "mygroup", NULL, 0);

    // 设置消费者的自动提交开关，默认为true
    rd_kafka_conf_set(conf, "enable.auto.commit", "true", NULL, 0);

    // 设置消费者的自动提交间隔（以毫秒为单位），默认为5000
    rd_kafka_conf_set(conf, "auto.commit.interval.ms", "5000", NULL, 0);

    // 创建消费者实例
    char errstr[512];
    rd_kafka_t *consumer = rd_kafka_new(RD_KAFKA_CONSUMER, conf, errstr, sizeof(err
    if (!consumer) {
        fprintf(stderr, "Failed to create consumer: %s\\n", errstr);
        return 1;
    }

    // 订阅主题
    rd_kafka_topic_partition_list_t *topics = rd_kafka_topic_partition_list_new(1);
    rd_kafka_topic_partition_list_add(topics, "test", RD_KAFKA_PARTITION_UA);
    if (rd_kafka_subscribe(consumer, topics) != RD_KAFKA_RESP_ERR_NO_ERROR) {
        fprintf(stderr, "Failed to subscribe to topic: %s\\n", rd_kafka_err2str(rd_
        rd_kafka_topic_partition_list_destroy(topics);
        rd_kafka_destroy(consumer);
        return 1;
    }
    rd_kafka_topic_partition_list_destroy(topics);

    // 消费消息
    while (1) {
        rd_kafka_message_t *rkmessage = rd_kafka_consumer_poll(consumer, 1000);
        if (rkmessage) {
```

```
            msg_consume(rkmessage, NULL);
            rd_kafka_message_destroy(rkmessage);
        }
    }

    // 取消订阅
    rd_kafka_unsubscribe(consumer);

    // 销毁消费者实例
    rd_kafka_destroy(consumer);

    return 0;
}
```

手动提交位点

手动提交位点：消费者在处理完消息后需要手动提交位点。这种方式的优点是可以精确控制位点的提交，避免消息重复消费或丢失。但是需要注意，手动提交位点如果太频繁会导致 Broker CPU 很高，影响性能，随着消息量增加，CPU 消费会很高，影响正常 Broker 的其他功能，因此建议间隔一定消息提交位点。



```
#include <stdio.h>
#include <string.h>
#include <librdkafka/rdkafka.h>

// 消费者消息处理回调
void msg_consume(rd_kafka_message_t *rkmessage, void *opaque) {
    if (rkmessage->err) {
        fprintf(stderr, "%% Consume error for topic \\"%s\\" [%"PRId32"] "
                "offset %"PRId64": %s\\n",
                rd_kafka_topic_name(rkmessage->rkt),
                rkmessage->partition, rkmessage->offset,
```

```
                rd_kafka_message_errstr(rkmessage));
    } else {
        printf("%% Message received on topic %s [%"PRId32"] at offset %"PRId64": %.
               rd_kafka_topic_name(rkmessage->rkt),
               rkmessage->partition,
               rkmessage->offset, (int)rkmessage->len, (const char *)rkmessage->pay
    }
}

int main() {
    rd_kafka_conf_t *conf = rd_kafka_conf_new();

    // 设置Kafka集群的地址
    rd_kafka_conf_set(conf, "bootstrap.servers", "localhost:9092", NULL, 0);

    // 设置消费组ID
    rd_kafka_conf_set(conf, "group.id", "mygroup", NULL, 0);

    // 关闭消费者的自动提交开关
    rd_kafka_conf_set(conf, "enable.auto.commit", "false", NULL, 0);

    // 创建消费者实例
    char errstr[512];
    rd_kafka_t *consumer = rd_kafka_new(RD_KAFKA_CONSUMER, conf, errstr, sizeof(err
    if (!consumer) {
        fprintf(stderr, "Failed to create consumer: %s\\n", errstr);
        return 1;
    }

    // 订阅主题
    rd_kafka_topic_partition_list_t *topics = rd_kafka_topic_partition_list_new(1);
    rd_kafka_topic_partition_list_add(topics, "test", RD_KAFKA_PARTITION_UA);
    if (rd_kafka_subscribe(consumer, topics) != RD_KAFKA_RESP_ERR_NO_ERROR) {
        fprintf(stderr, "Failed to subscribe to topic: %s\\n", rd_kafka_err2str(rd_
        rd_kafka_topic_partition_list_destroy(topics);
        rd_kafka_destroy(consumer);
        return 1;
    }
    rd_kafka_topic_partition_list_destroy(topics);

    // 消费消息并手动提交位点
    int message_count = 0;
    while (1) {
        rd_kafka_message_t *rkmessage = rd_kafka_consumer_poll(consumer, 1000);
        if (rkmessage) {
            msg_consume(rkmessage, NULL);
```

```
            // 每隔10条消息手动提交位点
            if (++message_count % 10 == 0) {
                if (rd_kafka_commit_message(consumer, rkmessage, 0) != RD_KAFKA_RES
                    fprintf(stderr, "Failed to commit offset for message: %s\\n", r
                } else {
                    printf("Offset %"PRId64" committed\\n", rkmessage->offset);
                }
            }

            rd_kafka_message_destroy(rkmessage);
        }
    }

    // 取消订阅
    rd_kafka_unsubscribe(consumer);

    // 销毁消费者实例
    rd_kafka_destroy(consumer);

    return 0;
}
```

tRpc Go SDK

最近更新时间：2024-07-04 16:00:59

背景

TDMQ CKafka 是一个分布式流处理平台，用于构建实时数据管道和流式应用程序。它提供了高吞吐量、低延迟、可伸缩性和容错性等特性。
本文着重介绍 tRpc-Go-Kafka 客户端的关键参数和实践教程，以及常见问题。

调优实践

tRPC-GO-Kafka 封装开源 Kakfa SDK，利用 tRPC-Go 拦截器等功能，接入 tRPC-Go 生态。因此实践教程参见 Sarama Go：

常见问题

生产者问题

1. 使用 CKafka 生产消息时，出现错误 `Message contents does not match its CRC` 。

```
err:type:framework, code:141, msg:kafka client transport SendMessage: kafka server: Message contents does not match its CRC.
```

插件默认启用了 gzip 压缩，在 target 上加上参数 `compression=none` 关闭压缩即可。

```
target: kafka://ip1:port1?compression=none
```

2. 生产时同一用户需要有序，如何配置？

客户端增加参数 `partitioner` ，可选 random（默认），roundrobin，hash（按 key 分区）。

```
target: kafka://ip1:port1?clientid=xxx&partitioner=hash
```

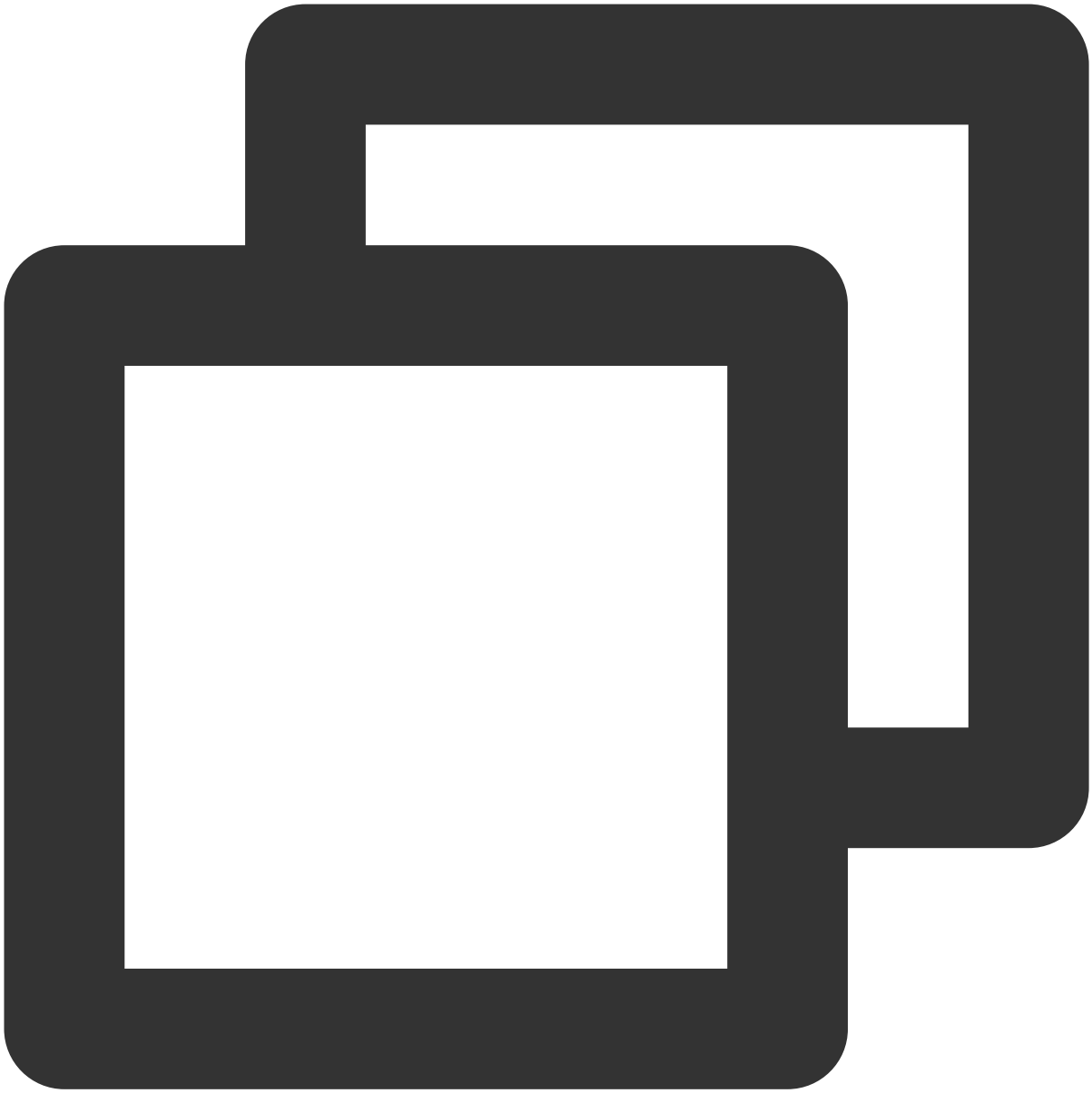
3. 如何异步生产？

客户端增加参数 `async=1`

```
target: kafka://ip1:port1,ip2:port2?clientid=xxx&async=1
```

4. 如何使用异步生产写数据回调？

需要在代码中重写异步生产写数据的成功/失败的回调函数，例如：



```
import (
  "git.code.oa.com/vicenteli/trpc-database/kafka"
)

func init() {
  // 重写默认的异步生产写数据错误回调
  kafka.AsyncProducerErrorCallback = func(err error, topic string, key, value []byt
    // do something if async producer occurred error.
  }

  // 重写默认的异步生产写数据成功回调
```

```
    kafka.AsyncProducerSuccCallback = func(topic string, key, value []byte, headers [
      // do something if async producer succeed.
    }
  }
```

消费者问题

1. 如果消费时 Handle 返回了非 nil 会发生什么？

会休眠 3s 后重新消费，不建议这么做，因为返回错误会导致无限重试消费，失败的应该由业务做重试逻辑。

2. 使用 ckafka 消费消息时，出现错误 `client has run out of available brokers to talk to` 。

`kafka server transport: consume fail:kafka: client has run out of available brokers to talk to (Is your cluster reachable?)`

优先检查 brokers 是否可达，然后检查支持的 kafka 客户端版本，尝试在配置文件 address 中加上参数例如 version=0.10.2.0

`address: ip1:port1?topics=topic1&group=my-group&version=0.10.2.0`

3. 当多个生产者生产时，部分生产者建立连接失败会影响正常的生产者超时？

请更新至 v0.2.18。 低版本插件在创建生产者时先加锁，再建立连接，建立连接结束后释放锁。如果存在一部分异常生产者建立连接耗时很长，就会导致其他正常生产请求在获取生产者的时候加锁失败，最终超时。此行为在 v0.2.18 已经修复。

4. 消费消息时，提示 `The provider group protocol type is incompatible with the other members` 。

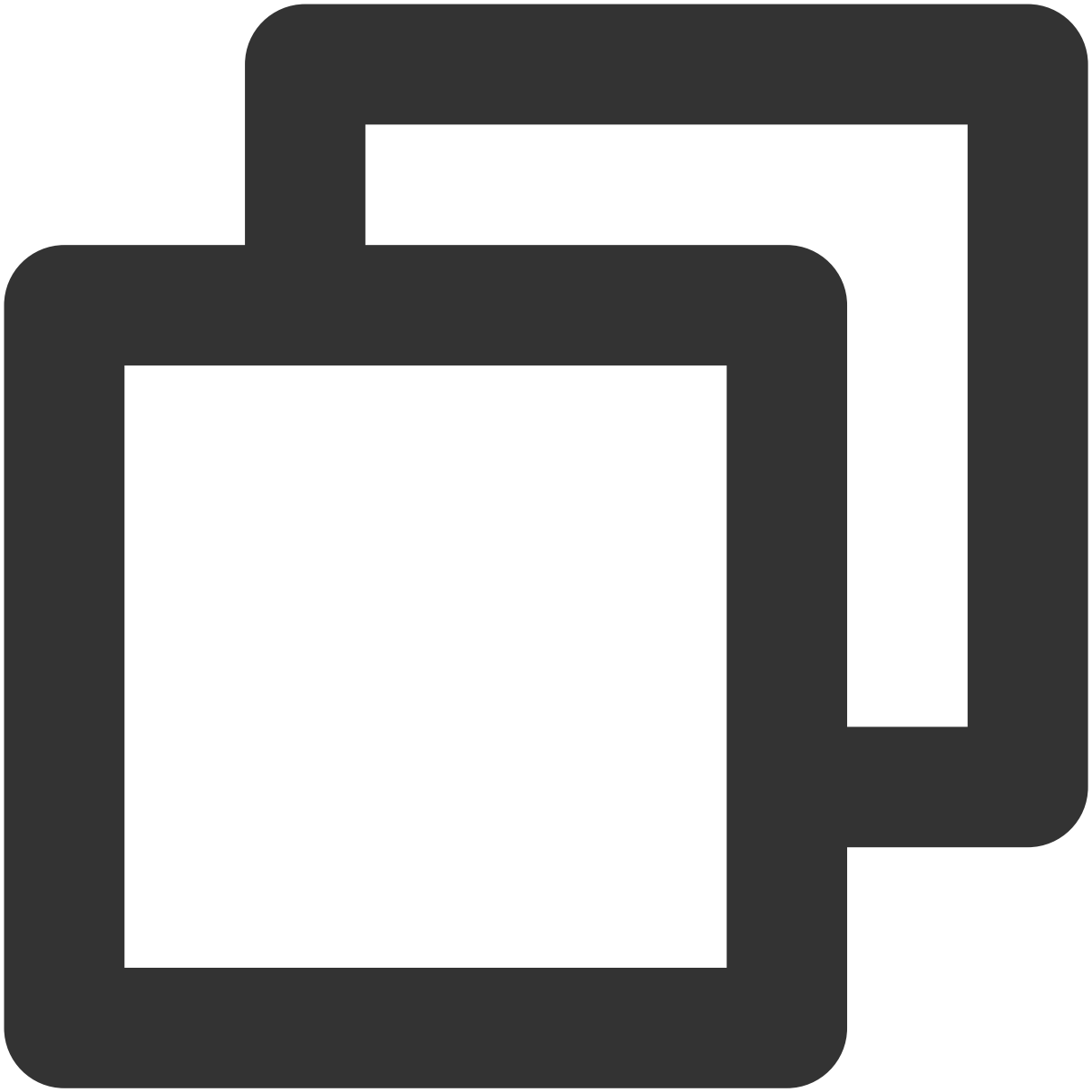
`kafka server transport: consume fail:kafka server: The provider group protocol type is incompatible with the other members.`

同一消费者组的客户端重分组策略不一样，可修改参数 `strategy` ，可选：sticky(默认)，range，roundrobin。

`address: ip1:port1?topics=topic12&group=my-group&strategy=range`

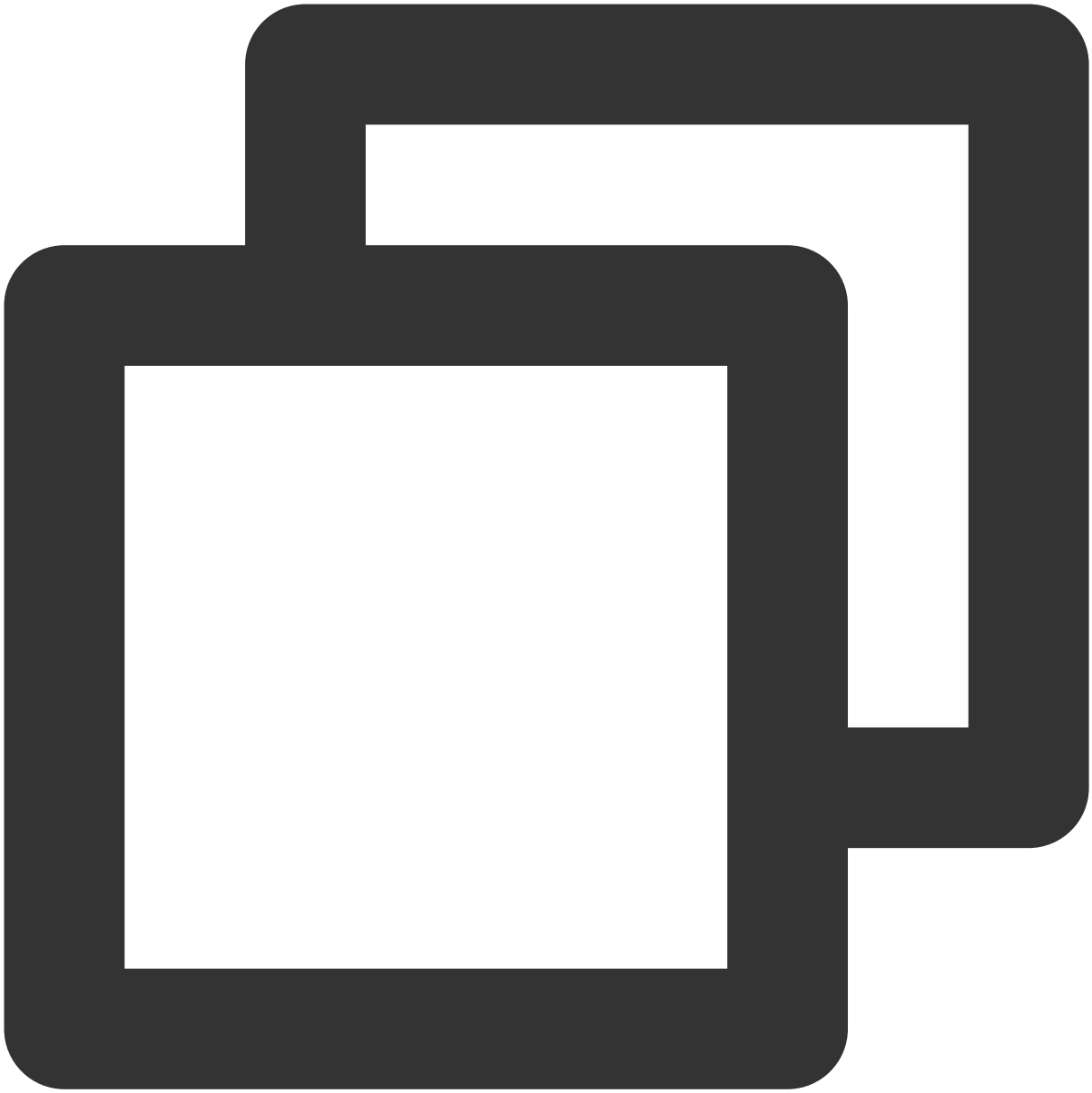
5. 如何注入自定义配置（远端配置）？

如果需要代码中指定配置，先在 `trpc_go.yaml` 中配置 `fake_address` ，然后配合 `kafka.RegisterAddrConfig` 方法注入， `trpc_go.yaml` 配置如下：



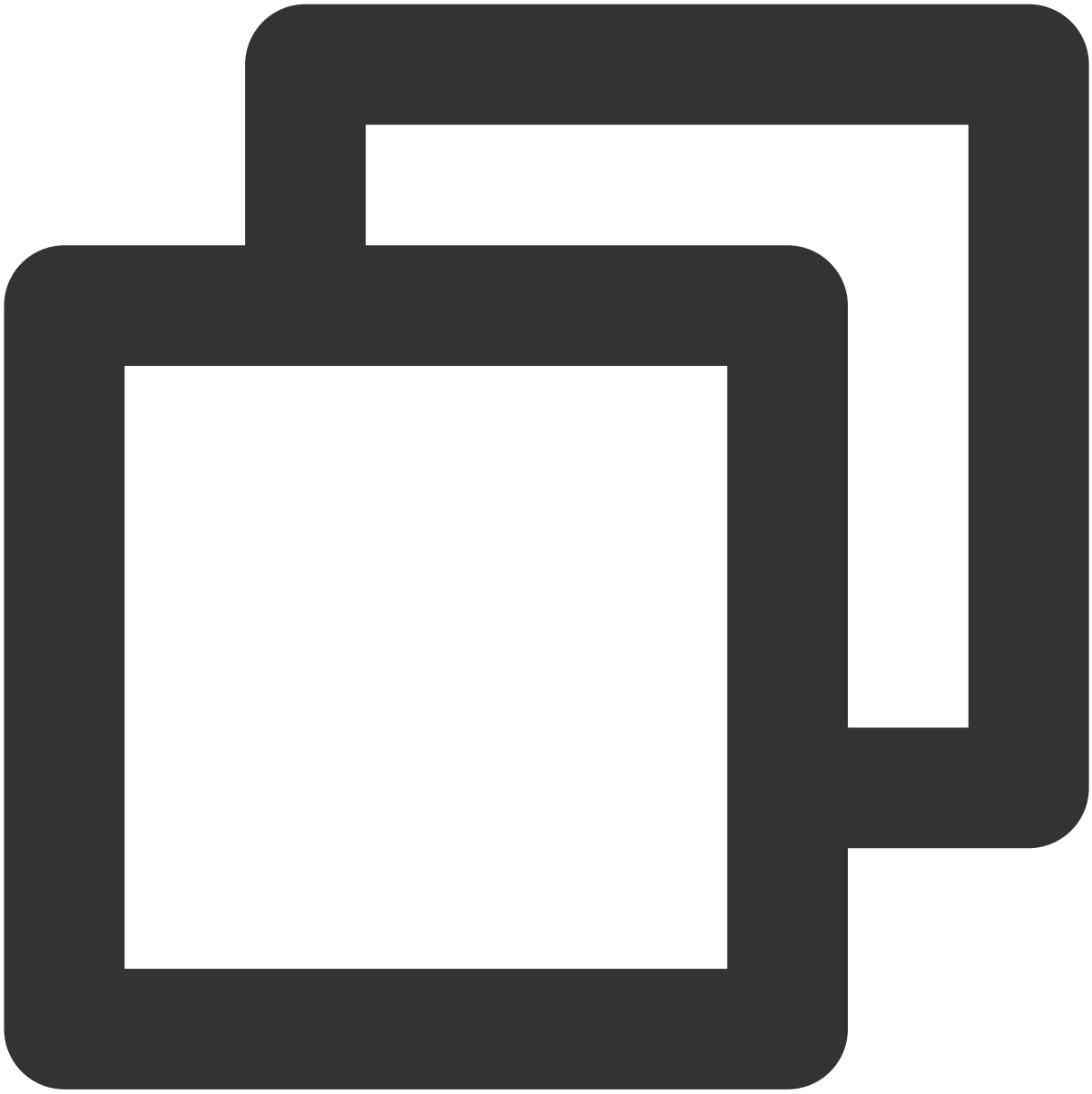
```
address: fake_address
```

在服务启动前，注入自定义配置：



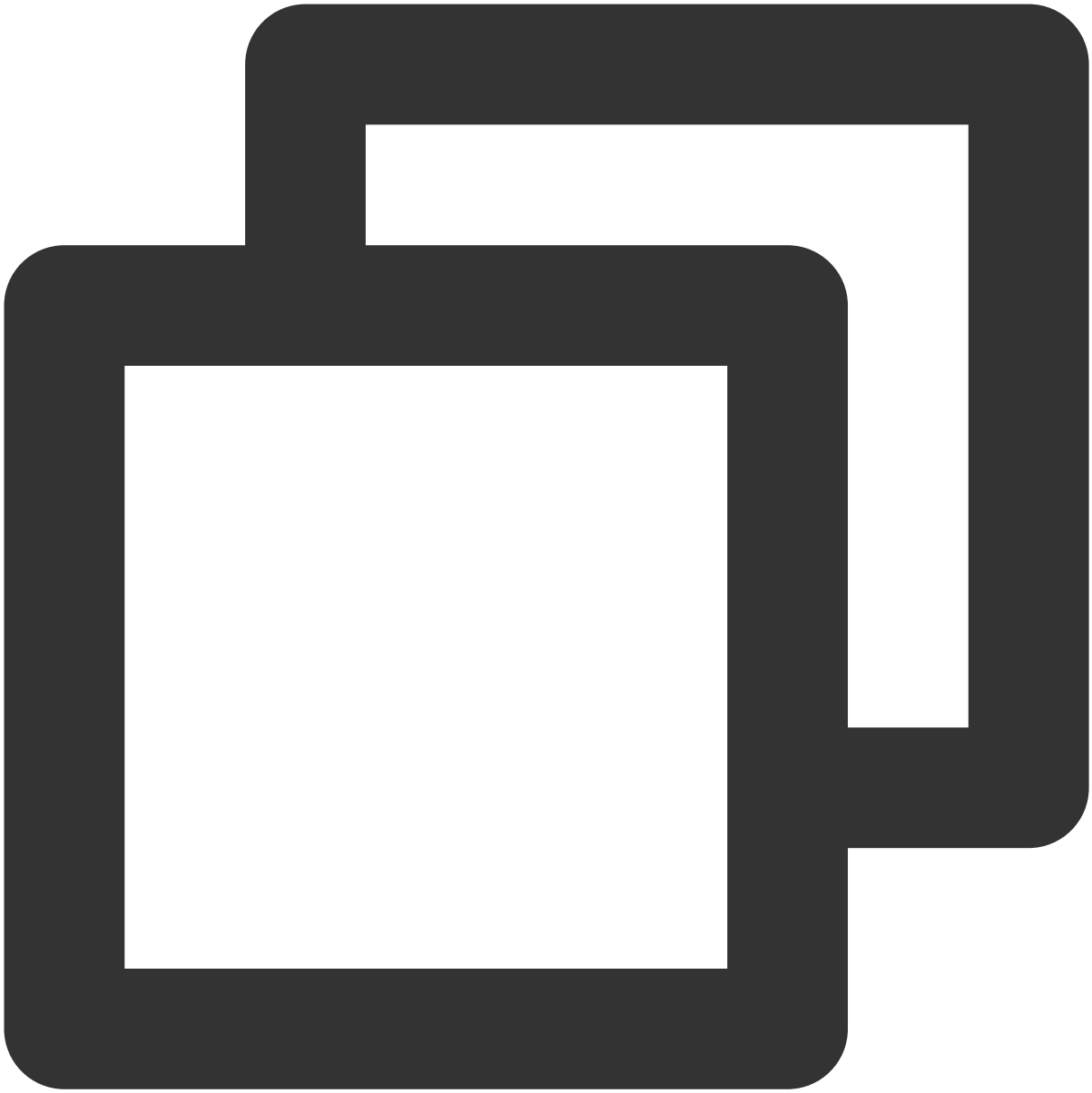
```
func main() {
  s := trpc.NewServer()
  // 使用自定义 addr，需在启动 server 前注入
  cfg := kafka.GetDefaultConfig()
  cfg.Brokers = []string{"127.0.0.1:9092"}
  cfg.Topics = []string{"test_topic"}
  kafka.RegisterAddrConfig("fake_address", cfg)
  kafka.RegisterKafkaConsumerService(s.Service("fake_address"), &Consumer{})

  s.Serve()
}
```

6. 如何获取底层 sarama 的上下文信息?



```
通过 kafka.GetRawSaramaContext 可以获取底层 sarama ConsumerGroupSession 和 ConsumerGro
// RawSaramaContext 存放 sarama ConsumerGroupSession 和 ConsumerGroupClaim
// 导出此结构体是为了方便用户实现监控，提供的内容仅用于读，调用任何写方法属于未定义行为
type RawSaramaContext struct {
    Session  sarama.ConsumerGroupSession
    Claim sarama.ConsumerGroupClaim
}
```

使用实例：



```
func (Consumer) Handle(ctx context.Context, msg *sarama.ConsumerMessage) error {
    if rawContext, ok := kafka.GetRawSaramaContext(ctx); ok {
        log.Infof("InitialOffset: %d", rawContext.Claim.InitialOffset())
    }
    // ...
    return nil
}
```

连接器实践教程
HTTP 上报
HTTP 协议接入 Kafka

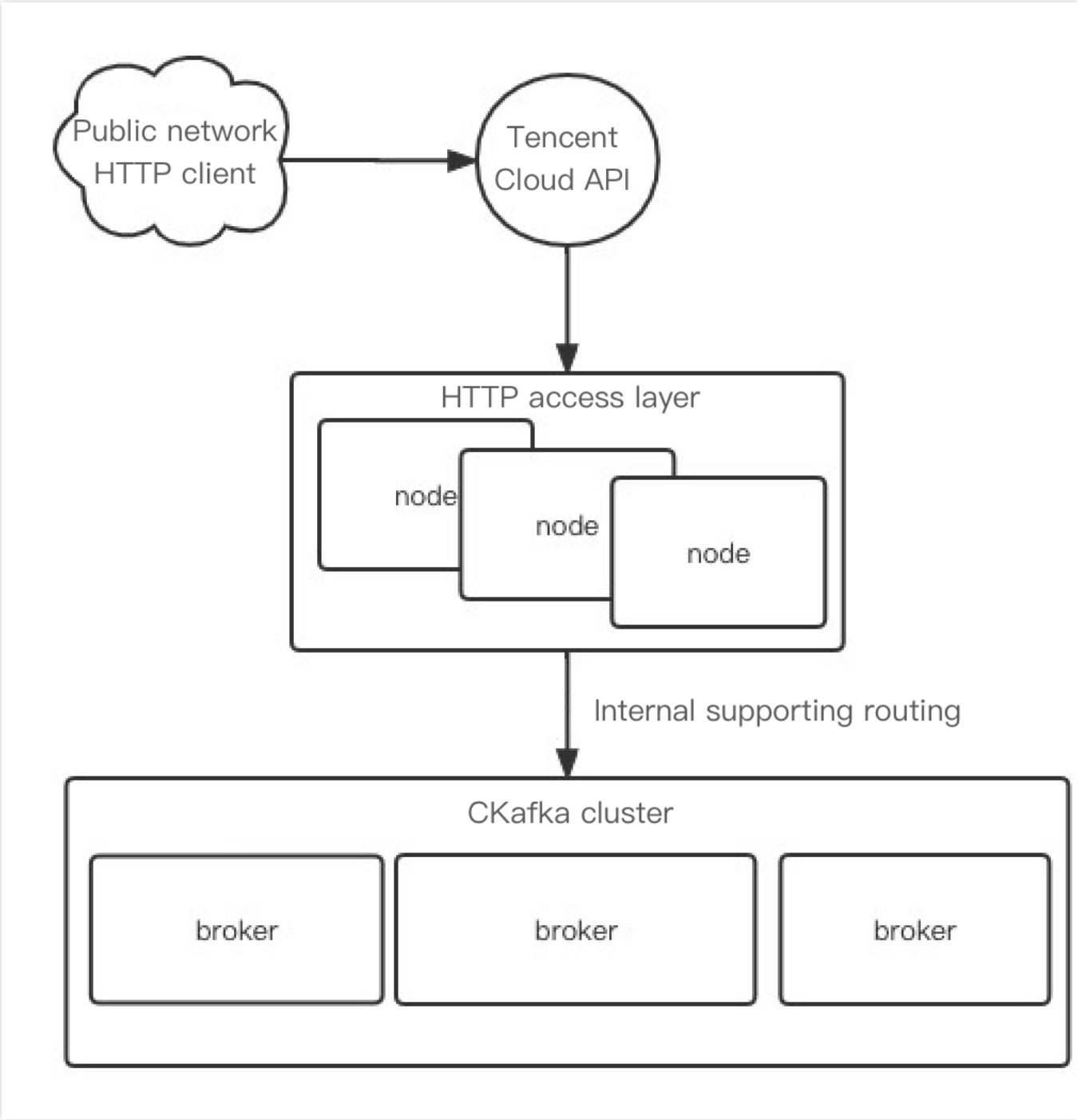
最近更新时间：2024-01-09 14:56:36

操作场景

与任何客户端-服务器应用程序一样，Kafka 通过一组明确定义的 API 提供对其功能的访问，这些 API 通过 Kafka 协议公开，是一种仅限于 Kafka 的 TCP 二进制协议。与 Kafka API 交互的最佳方式是客户端通过使用 Kafka 协议，Apache Kafka 项目仅正式支持 Java 的客户端库，但除此之外，Confluent 还正式支持 C/C++，C#，Go 和 Python 的客户端库。
一些编程语言缺乏官方支持的 Kafka 生产级客户端，而 HTTP 是一种广泛可用、普遍支持的协议，DataHub 数据接入通过 HTTP 协议公开消息发送 API，以便于简化客户端复杂的配置。
本文介绍 DataHub 服务的 HTTP 数据接入功能中的发送消息结合实际应用场景提供建议。

技术架构

HTTP 数据接入层开启后，公网的 HTTP 客户端可通过云 API 直接向 CKafka 所在的实例发送消息。示意图如下：



前提条件

已创建好目标 CKafka 实例和 Topic。

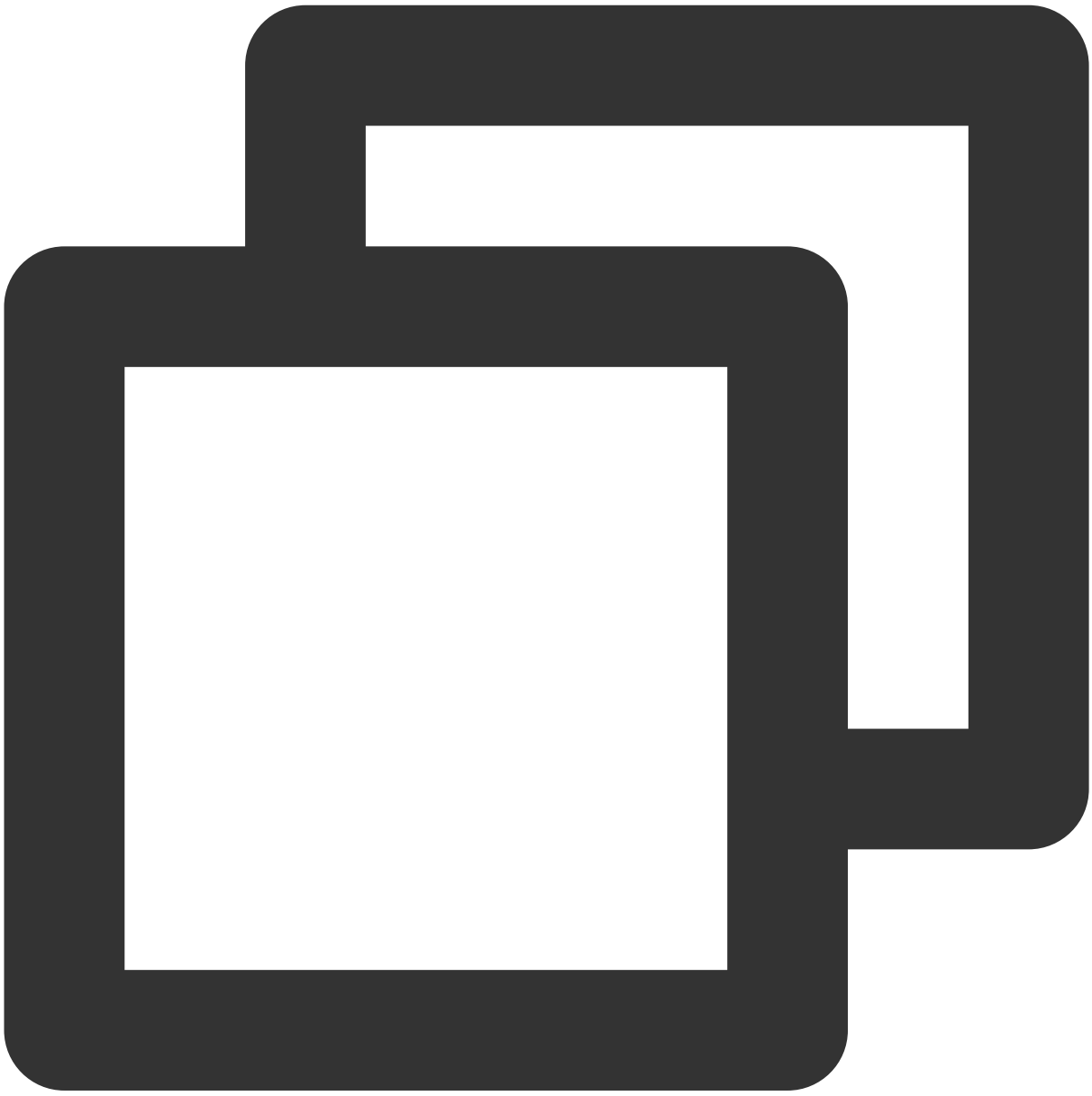
操作步骤

创建数据接入任务

具体操作请参见 DataHub 操作指南-HTTP 主动上报。

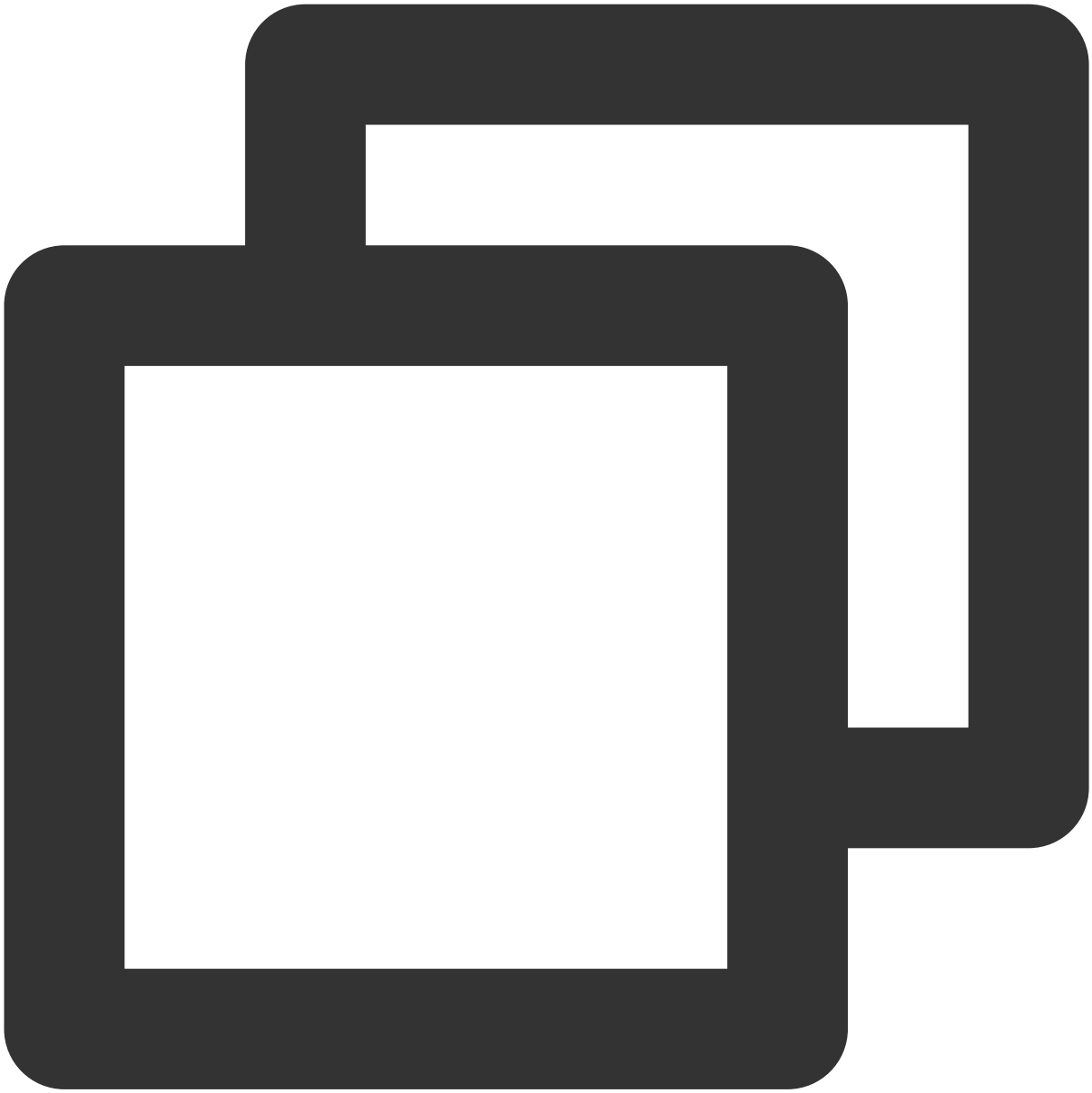
使用 SDK 发送消息

1. 在 Java 项目通过 Maven、Gradle 等方式引入数据上报 SDK。以下是配置项目的 pom.xml 文件。



```
<dependency>
    <groupId>com.tencentcloudapi</groupId>
    <artifactId>tencentcloud-sdk-java</artifactId>
    <version>3.1.430</version>
</dependency>
```

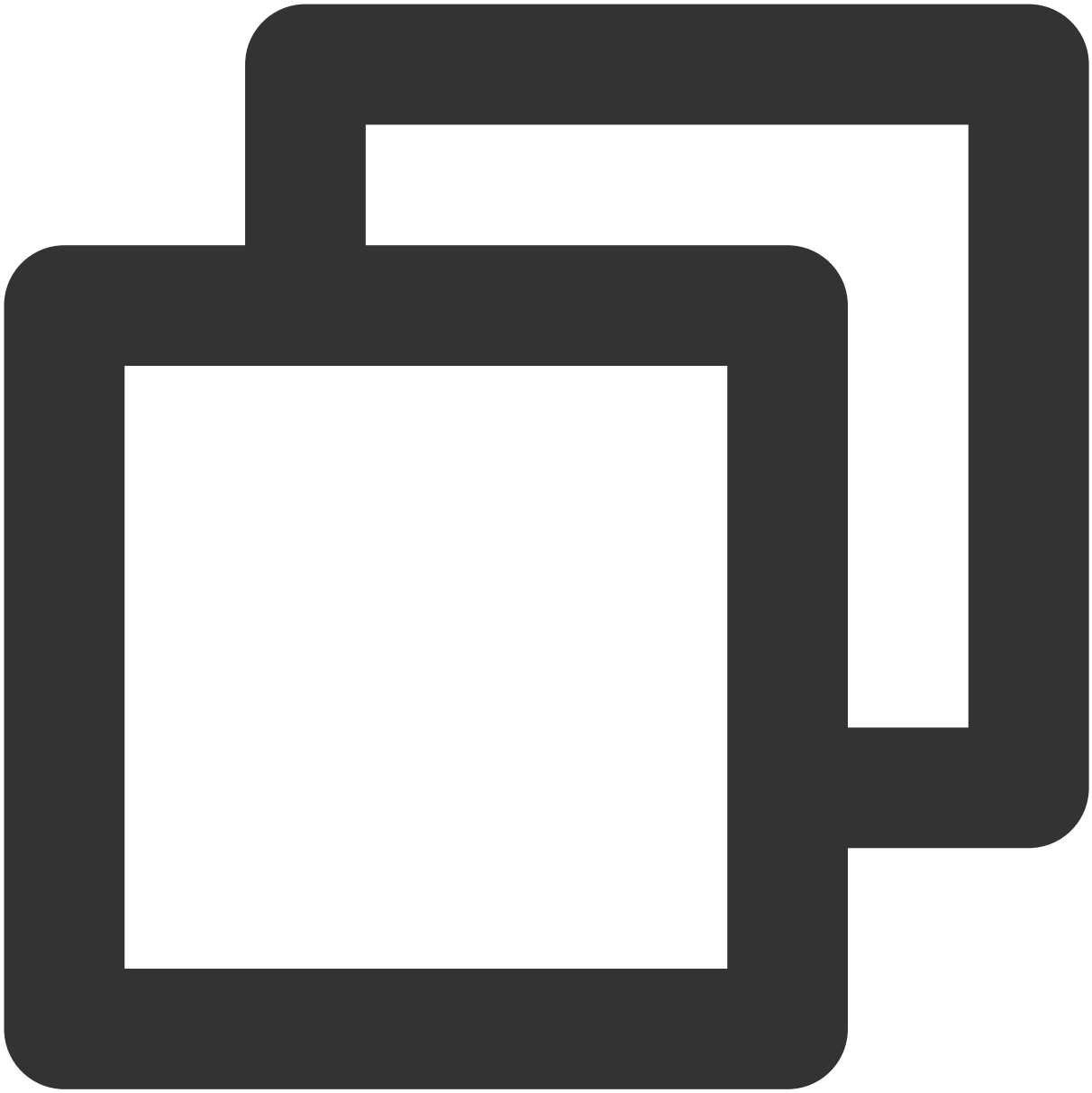
2. 单击 数据接入 的任务详情， 复制接入点信息到 SDK 中使用，用于写入数据。
3. 示例中通过 **generateMsgFromUserAccess** 将所有要发送的消息组装起来，复制接入点信息。



```
    List<BatchContent> batchContentList = generateMsgFromUserAccess(userId);
    // 其中 ap-xxx 为对应的云API地域简称
    CkafkaClient client = new CkafkaClient(
    new Credential("yourSecretId", "yourSecretKey"), "ap-xxx");


SendMessageRequest messageRequest = new SendMessageRequest();
```

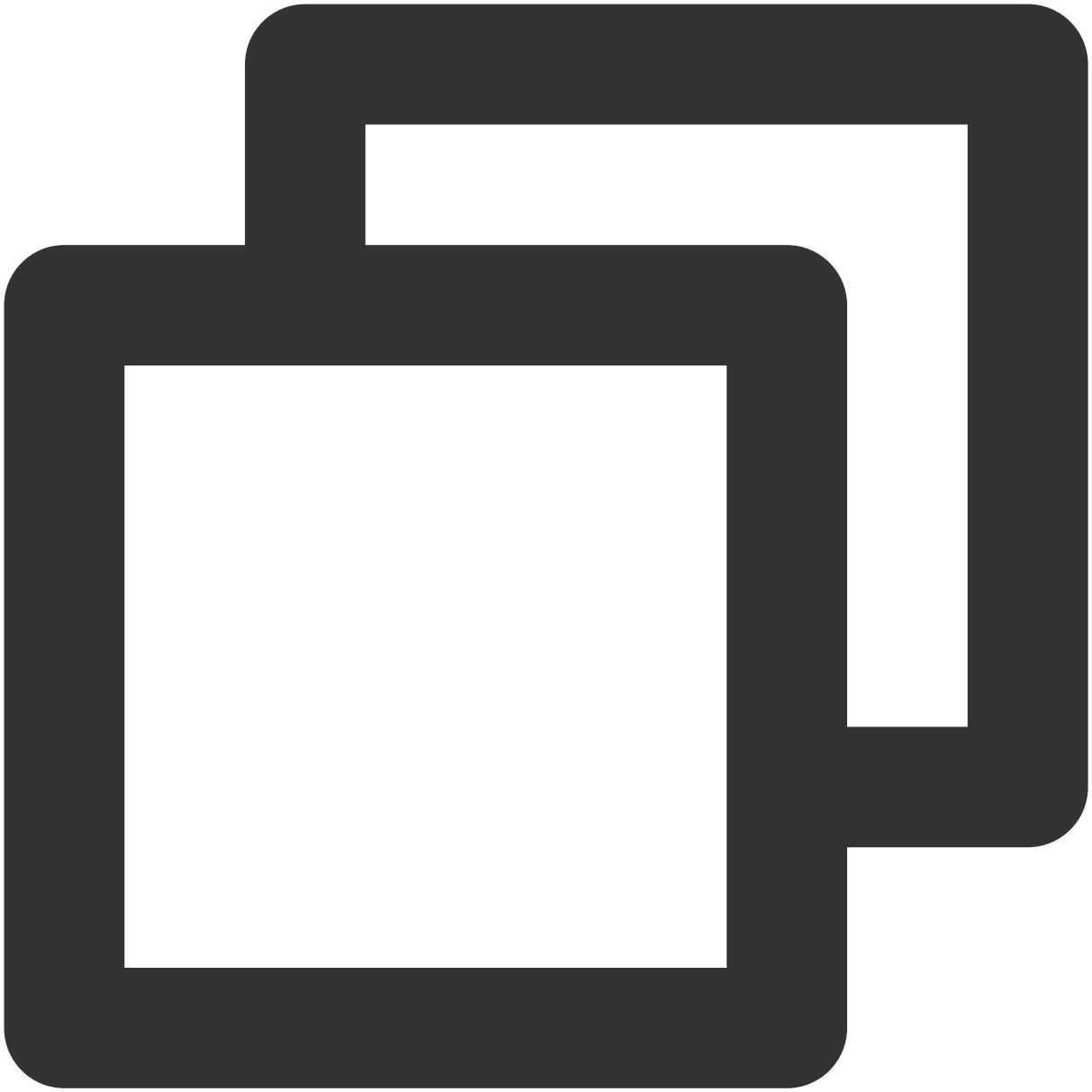
```
// 数据接入任务接入点ID
messageRequest.setDataHubId("datahub-lzxxxxx6");
messageRequest.setMessage(batchContentList.toArray(BatchContent[]::new));

try {
  SendMessageResponse sendMessageResponse = client.SendMessage(messageRequest);
  String[] messageId = sendMessageResponse.getMessageId();
  for (String s : messageId) {
     LOGGER.info(s)
  }
} catch (TencentCloudSDKException e) {
  LOGGER.error(e.getMessage());
}
```

4. 通过 HTTP 接入层发送消息的返回值示例如下。



```
{
    "Response": {
        "MessageId": [
            "datahub-lxxxxxx6:topicDev:4:2:1648185961342:1648185961398"
        ],
        "RequestId": "3fq3na5r-xxxx-xxxx-xxxx-b2fiv0se7ded"
    }
}
```

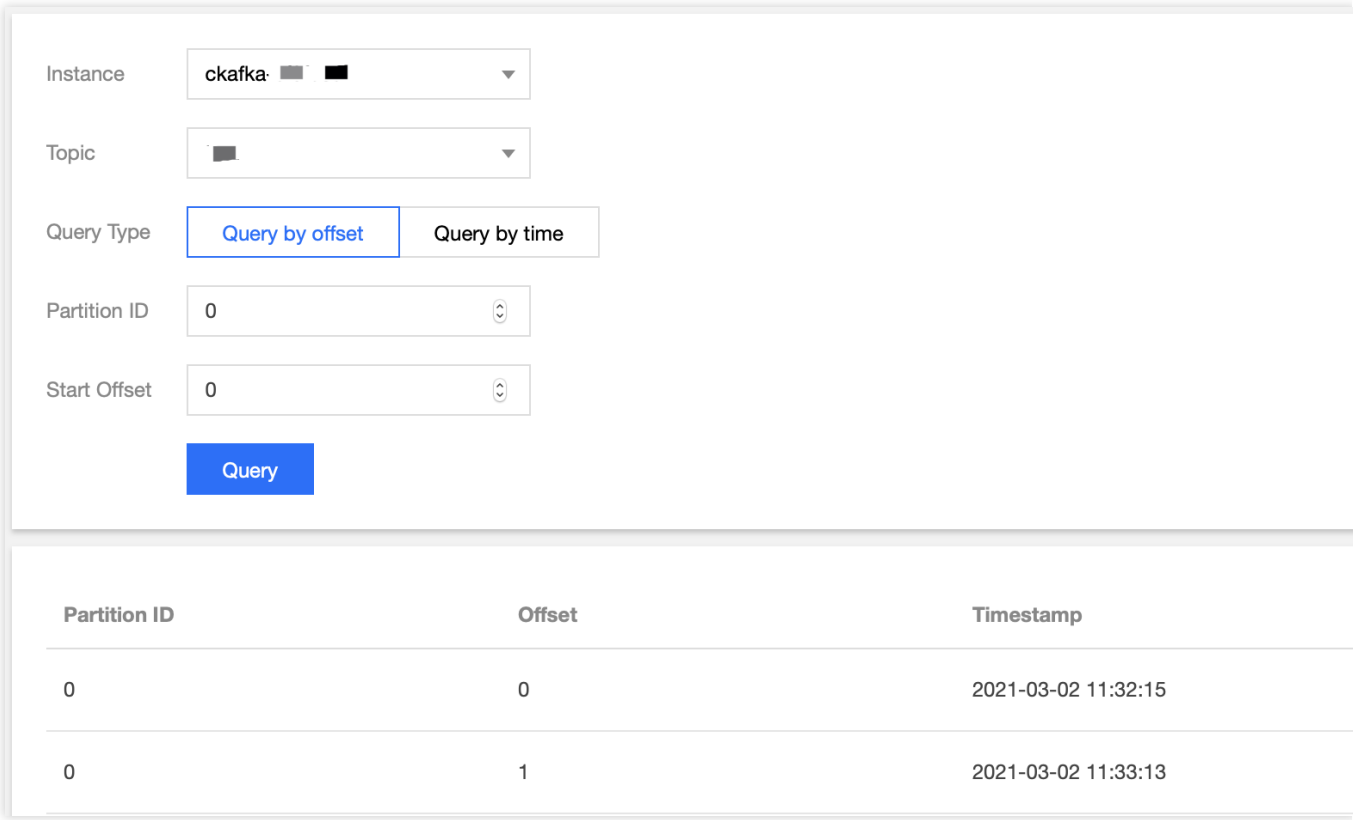
5. 其中 **MessageId** 内容由一系列发送至 CKafka 实例后返回的元数据组成。如下分别为：



```
"[datahubId]:[topic名称]:[所在的topic分区数]:[所在分区的offset]:[HTTP接入层收到消息的时间]:
```

查询消息

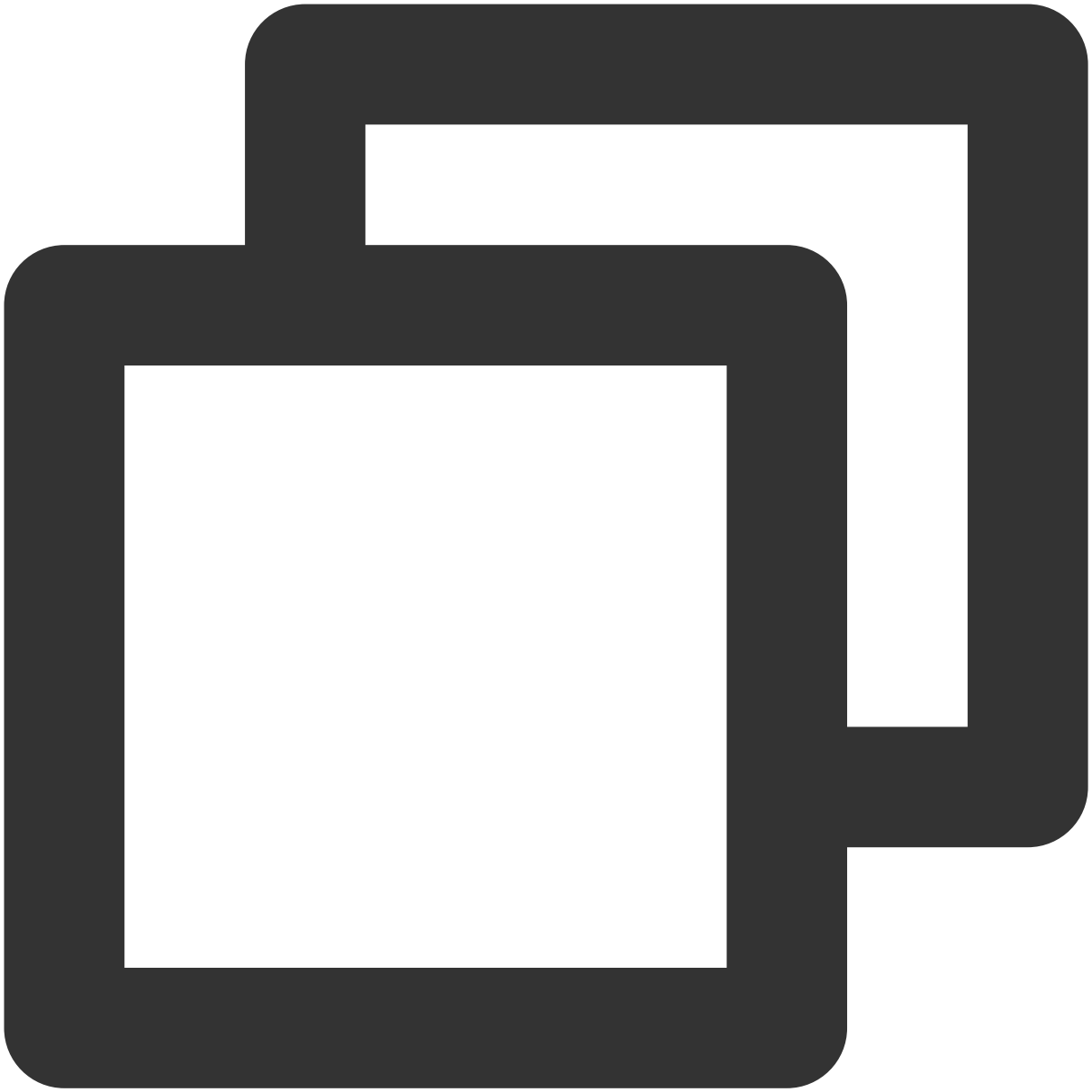
通过 CKafka 控制台 查询 HTTP 接入层发送的消息，详细操作参见 消息查询。如下图，示例 topic 名称为 topicDev 的4号分区查询2号位点消息。



任务暂停

当您发现数据接入任务影响了正常业务时，可以暂停数据接入。

1. 在 数据接入 页面，单击目标任务的操作栏的**暂停**，可暂停任务。
2. 出现右上角的提示，则任务暂停成功。
3. 此时通过 HTTP 接入层发送消息得到示例如下：



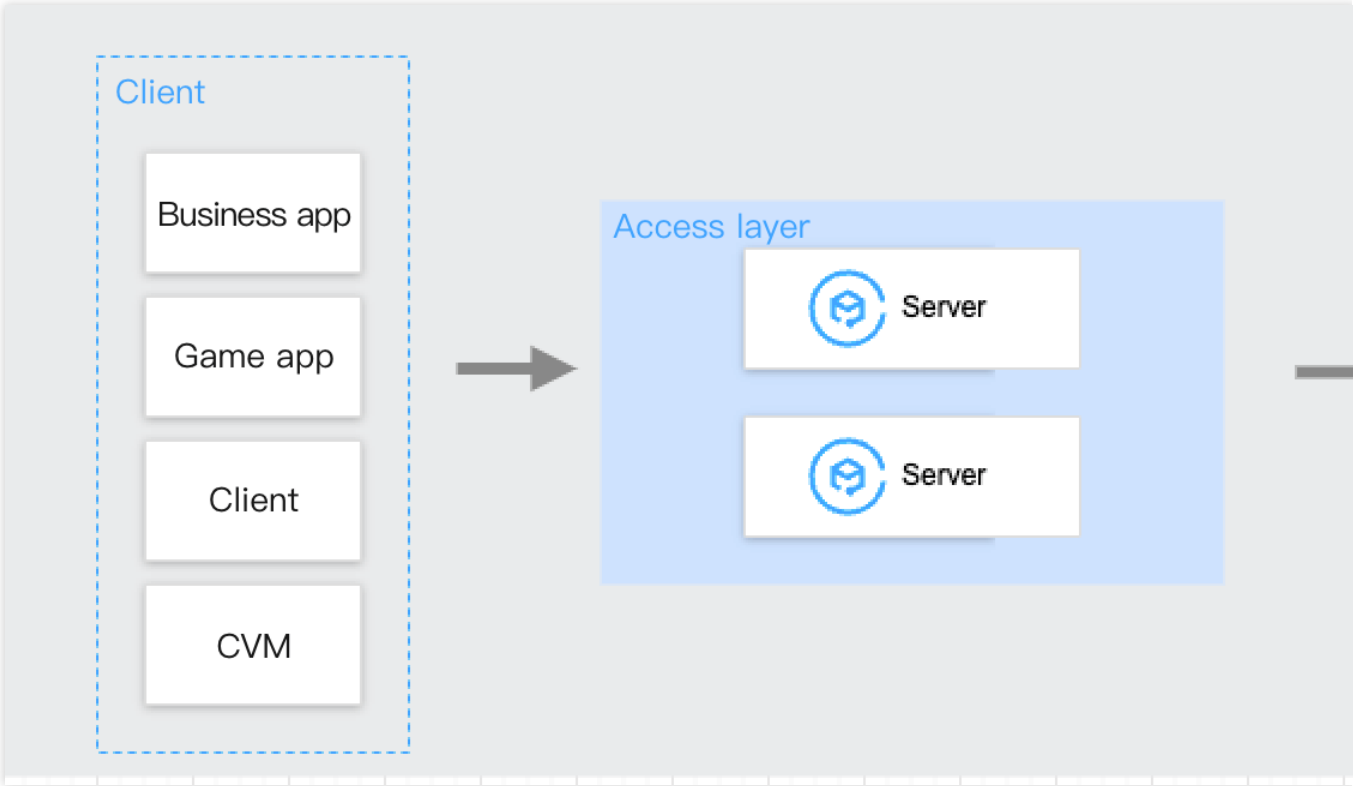
```
{
    "Response": {
        "Error": {
            "Code": "FailedOperation",
            "Message": "task status suspended [datahub-lxxxxxx6]"
        },
        "RequestId": "5f737a5b-xxxx-xxxx-xxxxx-b2fb703e7ded"
    }
}
```

统一数据上报

最近更新时间：2024-01-09 14:56:36

操作场景

数据接入平台（DataHub）是腾讯云上的数据接入和处理平台，一站式提供对数据的接入、处理和分发功能。数据在互联网业务中至关重要，而数据接入上报是整个链路中，介于数据产生和计算、存储、分析的桥梁，简单高效的数据接入是至关重要的。我们在业务经常有一些数据(业务指标、流程信息、监控信息等)需要上报到后台进行存储、分析、计算、搜索。其处理链路通常如下：



在经典数据上报架构中，通常需要有如下几个步骤：

1. 搭建/购买存储引擎，用来存储上报的数据。
2. 开发部署数据接收的 Server、定义 API、运行服务。
3. 定义客户端和服务端的接口协议、鉴权等信息
4. 客户端根据协议信息编写相应的代码完成数据上报。

在这四步中，Server 端的工作量是最大的，需要考虑代码逻辑开发、Server 本身扩缩容和稳定性、下游存储的扩缩容和稳定性等，而当数据量大时，Server 端问题会更加明显，需要消耗大量的人力物力来进行维护。但这部分工作通用性很高，DataHub 希望可以满足这种场景，提供稳定、弹性、高可靠、高吞吐的数据接入服务。

运行原理

DataHub 提供了 Java、Python、GoLang、PHP、NodeJs、C++、.Net 等各语言的 SDK 来方便客户端进行数据上报。通过3步即可将数据上报到存储引擎中，例如 Kafka（后续会支持 RocketMQ，Pulsar，RabbitMQ，CMQ 等多种消息队列）。具体步骤如下：

1. 在 DataHub 控制台创建接入点。
2. 通过 SDK 上报数据。
3. 数据查询。

操作步骤

1. 在 DataHub 控制台创建接入点。可参见 HTTP上报 创建接入点。
2. 通过 SDK 进行数据上报。使用步骤可参见 数据上报 SDK。
3. 数据查询。当数据上报到 DataHub 后，可以实时查询消息内容，详情请参见 消息查询。

数据赋能

当通过 DataHub 简单快速的完成数据接入后，如何让数据产生价值才是最重要的事情。DataHub 提供的两个核心功能：

数据处理

DataHub 提供了简单的数据 ETL 引擎，提供了可以满足大部分数据清洗需求的处理引擎，对数据进行简单的格式化和处理，以便进行后续的使用。

数据流出

当完成数据处理后，DataHub 可以满足以下多种场景的数据处理需求：

实时搜索： 当需要对数据进行搜索时，可以将数据流式实时导出到：Elasticsearch、CLS 等搜索服务进行实时搜索。

OLAP 分析：让需要对数据进行分析时，可以将数据流式实时导出到 ClickHouse、TDW 等引擎进行分析。

持久存储：当需要对数据进行持久存储时，可以将数据流式实时导出到 HDFS、COS 等持久化引擎进行存储。

流计算：当业务需要通过自定义代码处理数据时，可以通过标准的 Kafka 协议，使用 FLink、Spark、各语言代码对数据进行处理。

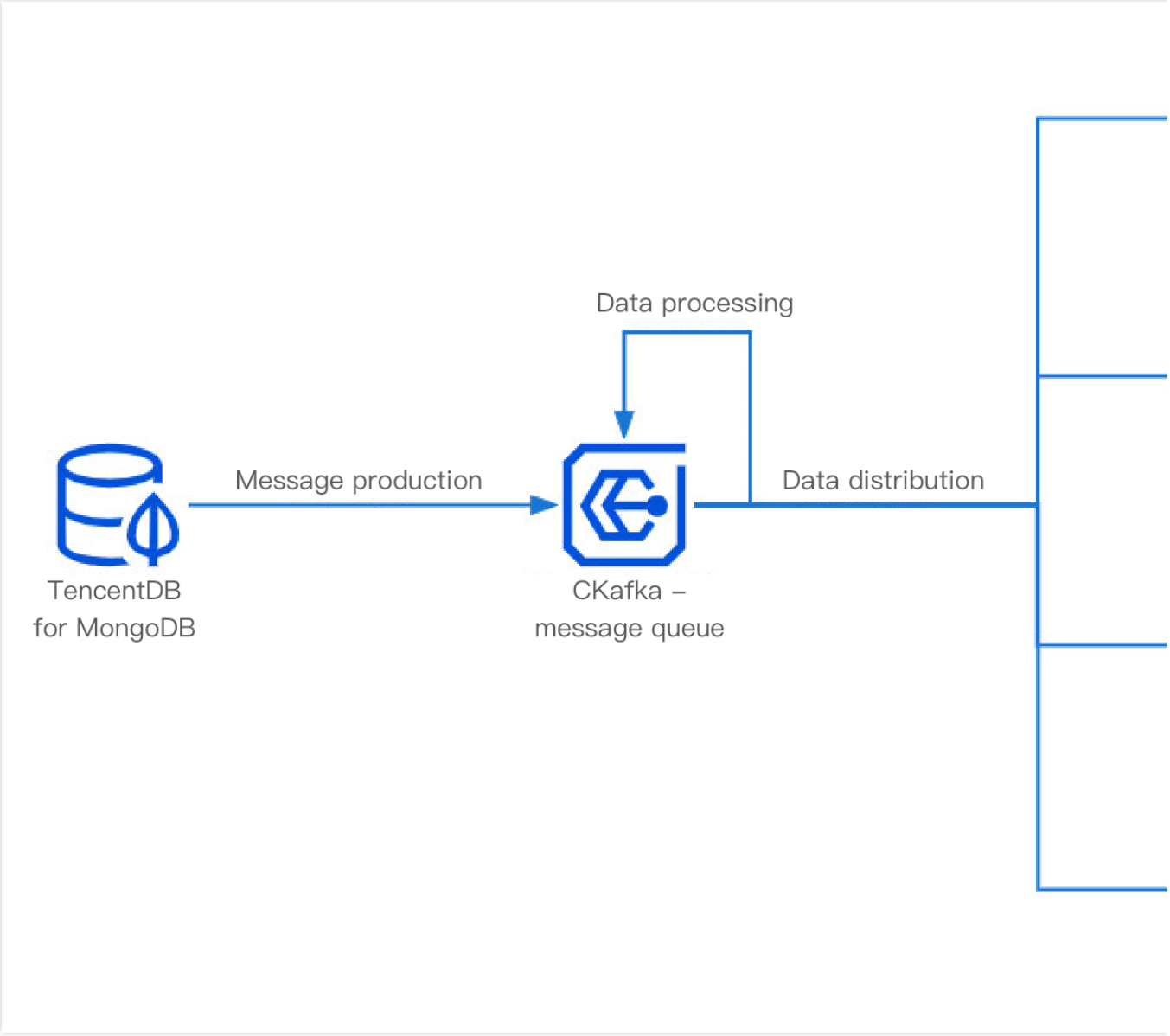
至此，数据经过数据上报、接入、处理、流出四个环节后，简单快速的满足通用的数据上报、分析类需求，用极低的成本让数据体现出价值。

数据库变更订阅查询
Mongo Stream 数据变更记录分析

最近更新时间：2024-01-09 14:56:36

操作场景

MongoDB 内有 Change Stream 作为其追踪变更的解决方案，但为了更好地对变更记录进行搜索，往往需要将变更记录同步到 Elasticsearch、日志服务(CLS) 等。



本文以 MongoDB 接入 CKafka 并从 CKafka 流出到 CLS 为例，讲解如何使用 DataHub 数据转储服务实现 Mongo Stream 数据变更记录分析。

运行原理

MongoDB 的数据流入配置项可参见官方 MongoDB Kafka source connector，通过设置不同的配置项，能够对接入的变更记录做相应的数据处理后再写入到 CKafka 实例的 Topic。

前提条件

需开启云数据库 MongoDB 服务或使用负载均衡 CLB 监听自建 MongoDB。
并且设置的安全组需要开放 TCP:27017 端口。
需开启 CKafka 服务。
需开启 CLS 服务。

操作步骤

步骤1：创建数据接入任务

1. 登录 CKafka 控制台。
2. 在左侧导航栏单击**数据流入**，选择好地域后，单击**新建任务**。
3. 在弹窗中数据源类型选择**异步拉取** > **MongoDB**。
4. 单击**下一步**，填写任务详情。
5. 单击**提交**，等待任务显示健康，即表示创建成功。
6. 当 MongoDB 数据发生变更时，可以看到选中的 CKafka 实例的 Topic 有新增的消息。

步骤2：创建数据流出任务

1. 登录 CKafka 控制台。
2. 在左侧导航栏单击**数据流出**，选择好地域后，单击**新建任务**。
3. 目标类型选择**日志服务（CLS）**，单击**下一步**。
4. 填写任务详情，选取与数据接入任务相同的 CKafka 实例和 Topic，保证在消息生产后能直接进行消费。
5. 单击**提交**，等待任务显示健康，即表示创建成功。

说明：
当任务在健康的状态时， Topic 有新增的消息写入，会直接被消费到指定的 CLS 日志主题中。

步骤3：查看流出数据

1. 登录 日志服务 控制台。
2. 在左侧导航栏选择**检索分析**，选择流出时填写的日志集与日志主题的“ID”，即可看到 MongoDB 的变更记录。
3. 通过关键字检索等操作，能直观得到所需要的记录。

数据简单清洗

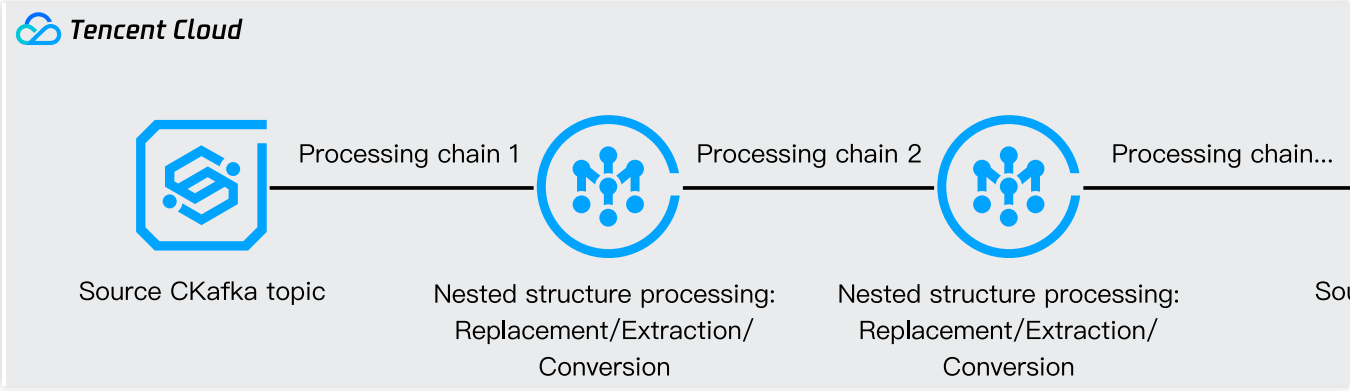
最近更新时间：2024-01-09 14:56:36

操作场景

在使用 DataHub 在进行数据流入流出服务时，可能会遇到如下情况：
需要对消息中的特定字段进行解析，得到相关信息。
需要多次迭代处理消息中某一字段。
需要处理为未经过结构化的原始消息后，才能继续使用。
需要处理多层嵌套格式的消息。
目前 CKafka 团队的技术专家经过长期的经验总结，全新推出了数据处理 V2.0 版本。 该版本在完美兼容原有数据处理 V1.0 的基础上，还增加了以下特性：
插件丰富：数据处理支持丰富的处理预制插件，利于快速生成所需的消费格式。
链式处理：数据处理支持消息链式处理，即上一次的处理结果能够作为本次处理的输入参数，利于处理复杂结构。
可视预览：数据处理支持实时 JSON 结构化预览，利于迅速定位排查问题。
类型转换：数据处理支持不同数据类型间的转换，利于校验数据格式。

运行原理

整个数据处理过程如下图所示，各个组件结构说明如下：



数据处理组件集群中，多个 worker 共同组成一个消费者组，批量读取源 topic 中的消息，并且对于每条消息顺序执行处理操作。
对于每条消息，根据控制台设置的处理链顺序，依次展开消息字段的嵌套结构，并进行替换/截取/数据转换/时间格式化等操作。
下一条处理链读取上一条处理链结果，并且进行本轮的链式处理，并将处理结果投递到下一轮。

执行完最后一轮处理链后的结果，投递到设置的目标 topic 中，至此完成该条消息的数据处理操作。

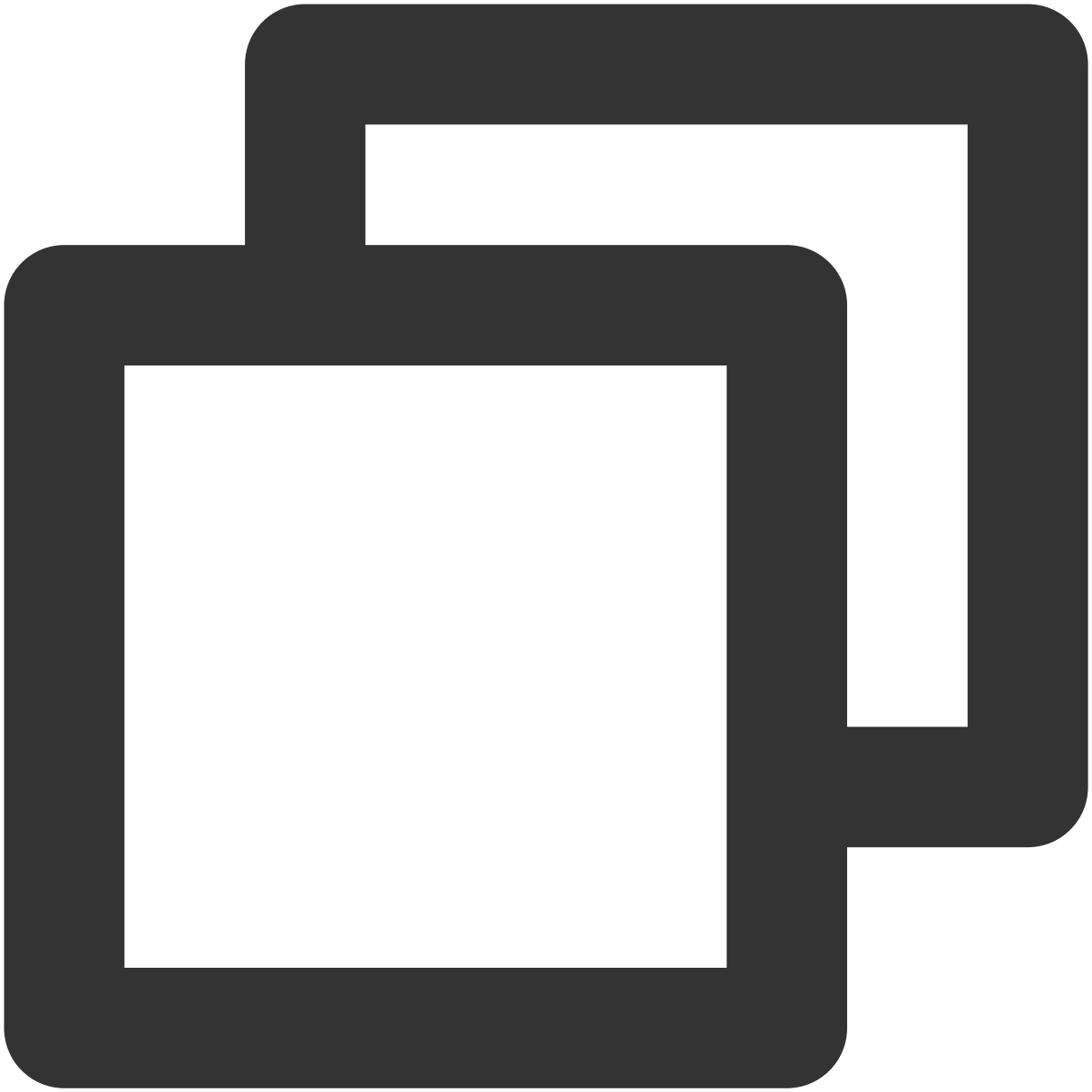
前提条件

需开通 CKafka 服务。
需消息格式为 JSON 格式或字符串格式，目前暂不支持其他编码协议。

实际应用

处理字符串类型格式日志

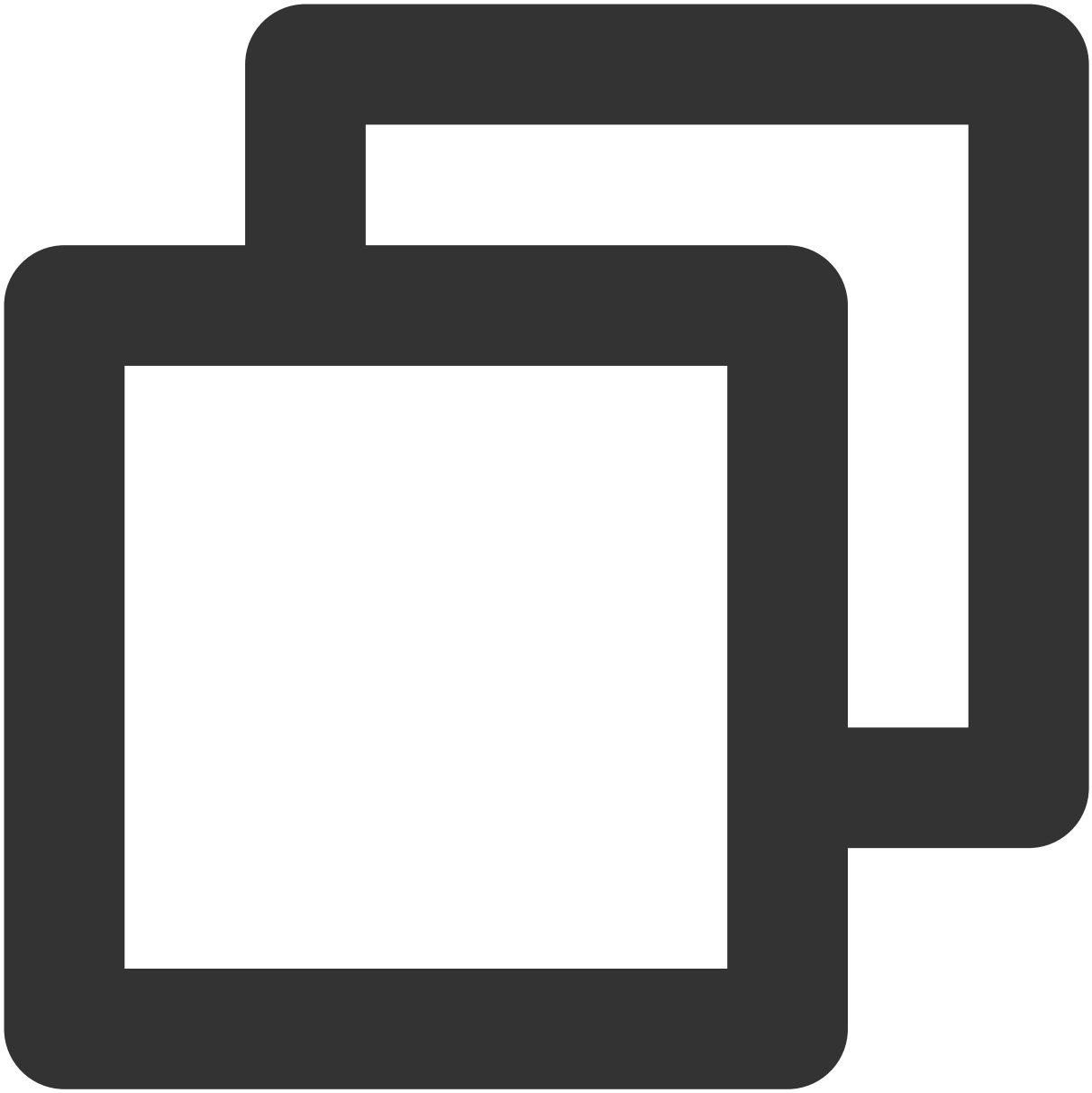
Nginx 格式日志通常如下所示，这种 Nginx 默认格式也被称作 `combined`，可以从中解析出请求者信息，报文信息，请求状态等信息，以便于对数据进行进一步分析。



```
66.249.65.159 - - [06/Nov/2014:19:10:38 +0600] "GET /news/53f8d72920ba2744fe873ebc.
```

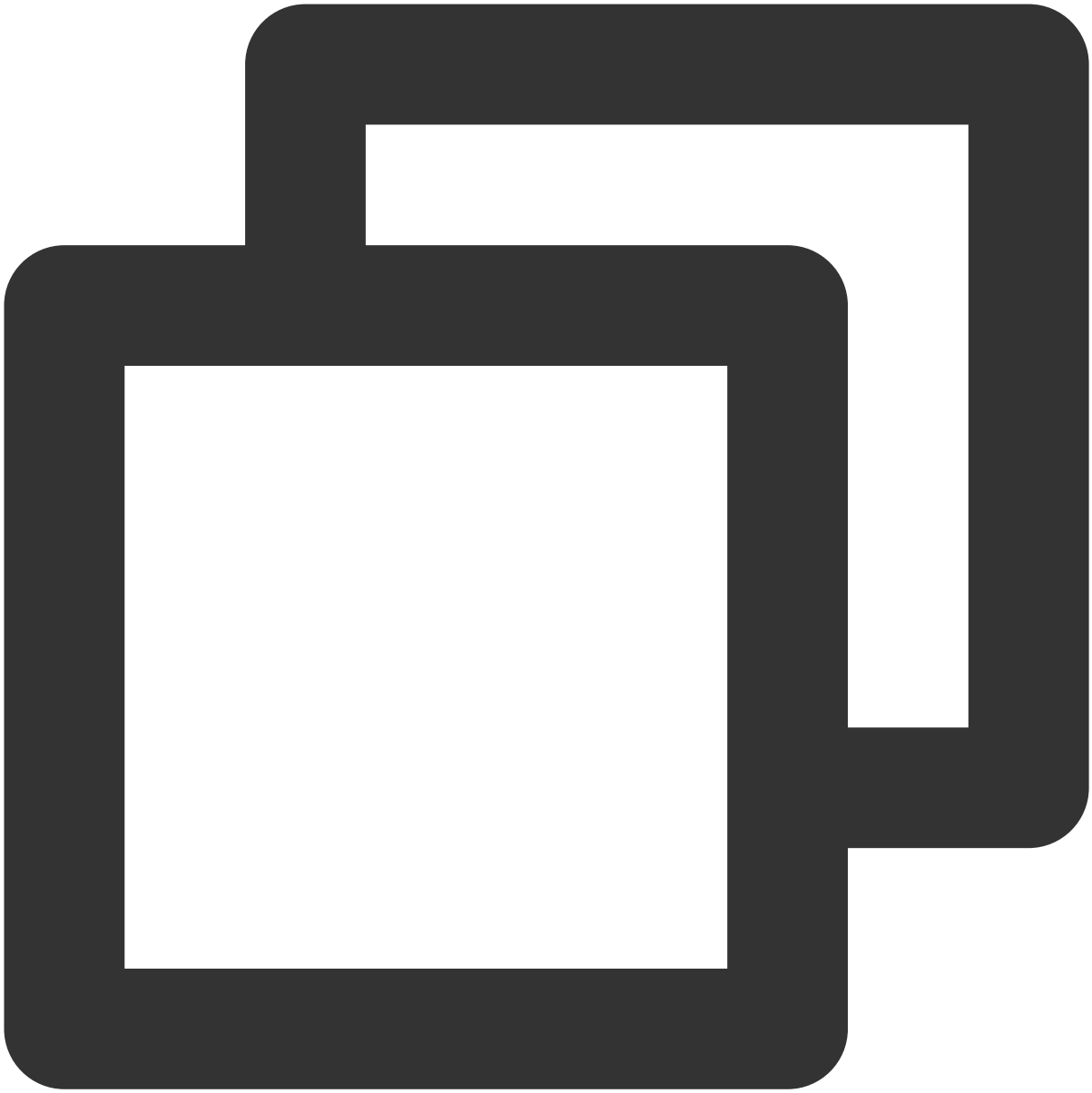
由于 Nginx 的 `combined` 格式采用空格和 `-` 分隔符来分隔数据，因此按照下文顺序设计解析流程：

1. 首先使用 `"` **分隔符** 进行初步数据解析，此时数据处理将会自动将日志转换成 JSON 结构



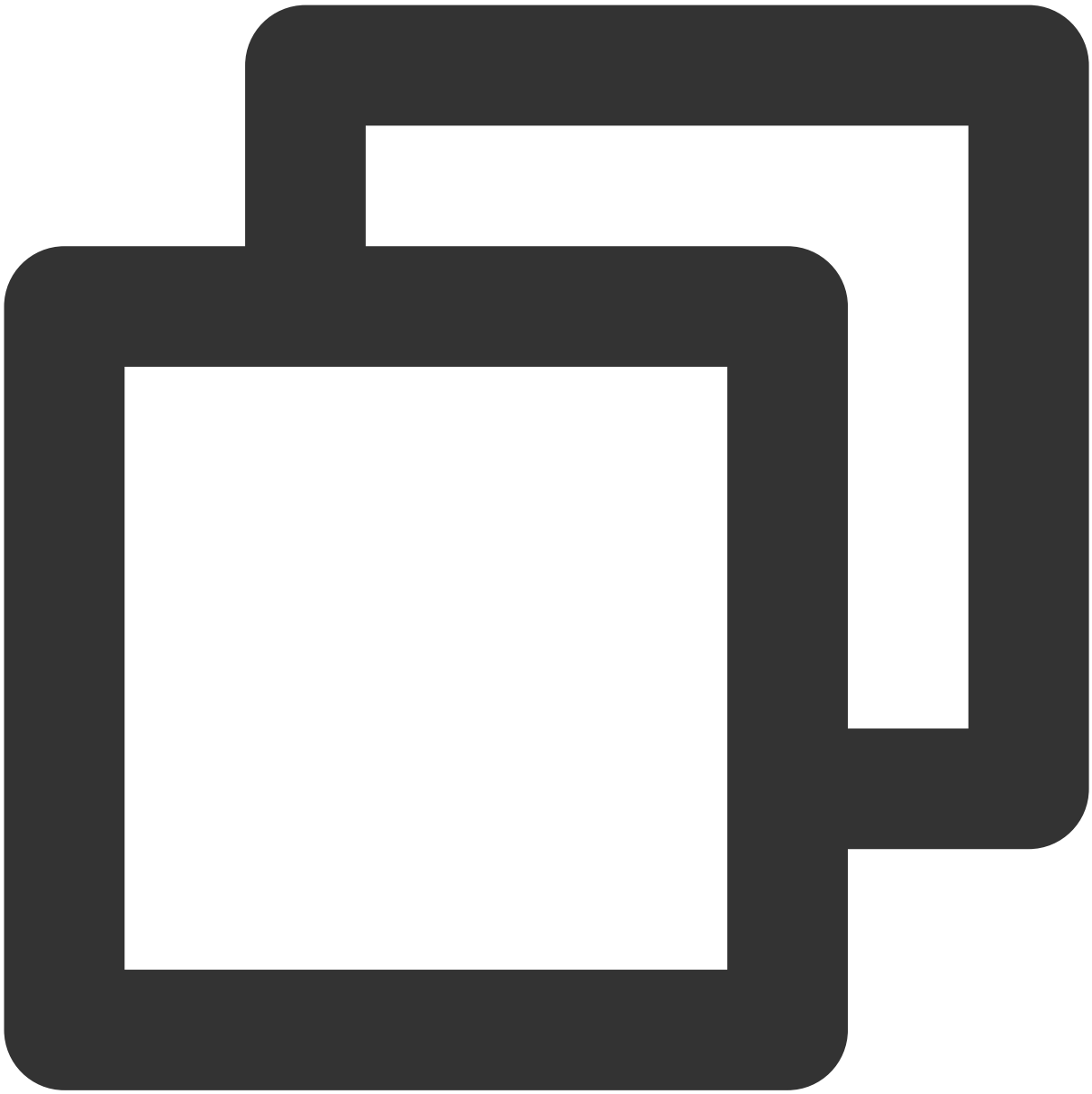
```
{
  "0": "66.249.65.159 - - [06/Nov/2014:19:10:38 +0600] ",
  "1": "GET /news/53f8d72920ba2744fe873ebc.html HTTP/1.1",
  "2": " 404 177 ",
  "5": "Mozilla/5.0 (iPhone; CPU iPhone OS 6_0 like Mac OS X) AppleWebKit/536.26 (K
}
```

2. 从上文 JSON 结构中可知，键名为 `0` 和 `2` 的字段由于 `-` 和空格的影响，还存在连接的耦合数据。因此再分别对这两个键进行 `-` 和空格进行**分隔符**拆分。 拆分后的 JSON 结果如下所示：



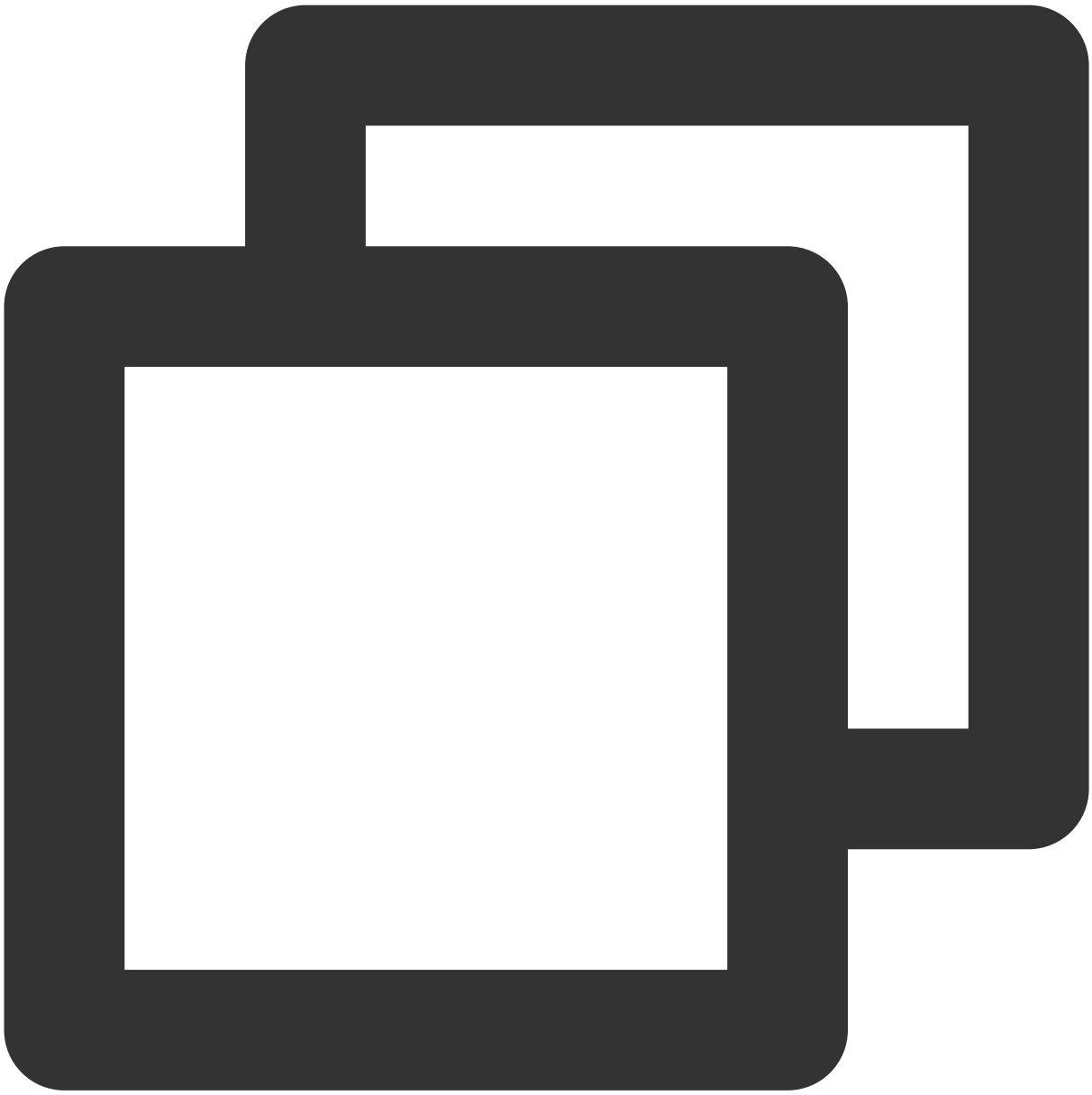

```
{
  "1": "GET /news/53f8d72920ba2744fe873ebc.html HTTP/1.1",
  "5": "Mozilla/5.0 (iPhone; CPU iPhone OS 6_0 like Mac OS X) AppleWebKit/536.26 (K
  "0.0": "66.249.65.159 ",
  "0.2": " [06/Nov/2014:19:10:38 +0600] ",
  "2.1": "404",
  "2.2": "177"
}
```

3. 由于时间格式前后还带有 `[]` 方括号，再使用一次**分隔符**进行截取，截取后的 JSON 如下所示：



```
{
  "1": "GET /news/53f8d72920ba2744fe873ebc.html HTTP/1.1",
  "5": "Mozilla/5.0 (iPhone; CPU iPhone OS 6_0 like Mac OS X) AppleWebKit/536.26 (K
  "0.0": "66.249.65.159 ",
  "0.2": "06/Nov/2014:19:10:38 +0600",
  "2.1": "404",
  "2.2": "177"
}
```

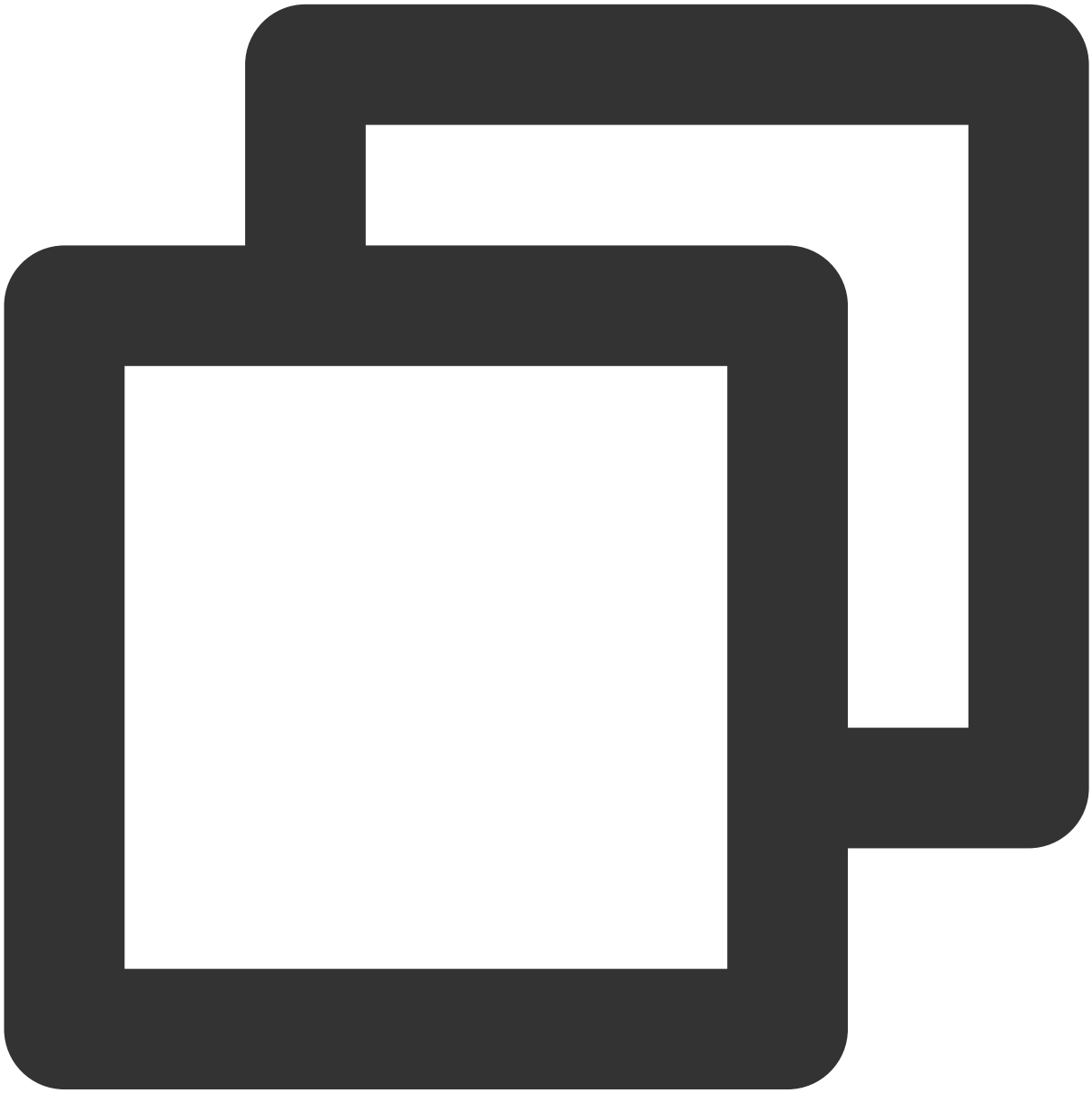
4. 由于所有字段都已经被合适拆分，最后根据对应字段属性，给相应映射字段的 `key` 设置名称即可。修改后最终结果如下所示：



```
{
  "request": "GET /news/53f8d72920ba2744fe873ebc.html HTTP/1.1",
  "http_user_agent": "Mozilla/5.0 (iPhone; CPU iPhone OS 6_0 like Mac OS X) AppleWe
  "remote_addr": "66.249.65.159 ",
  "dateTime": "06/Nov/2014:19:10:38 +0600",
  "status": "404",
  "body_bytes_sent ": "177"
}
```

处理嵌套类型格式日志

TKE 采集格式通常如下所示，TKE 采集器会将 metadata 数据放入 JSON 结构中的 `kubernetes` 字段中，采集到的日志放入 `log` 字段中。大致结构如下所示：

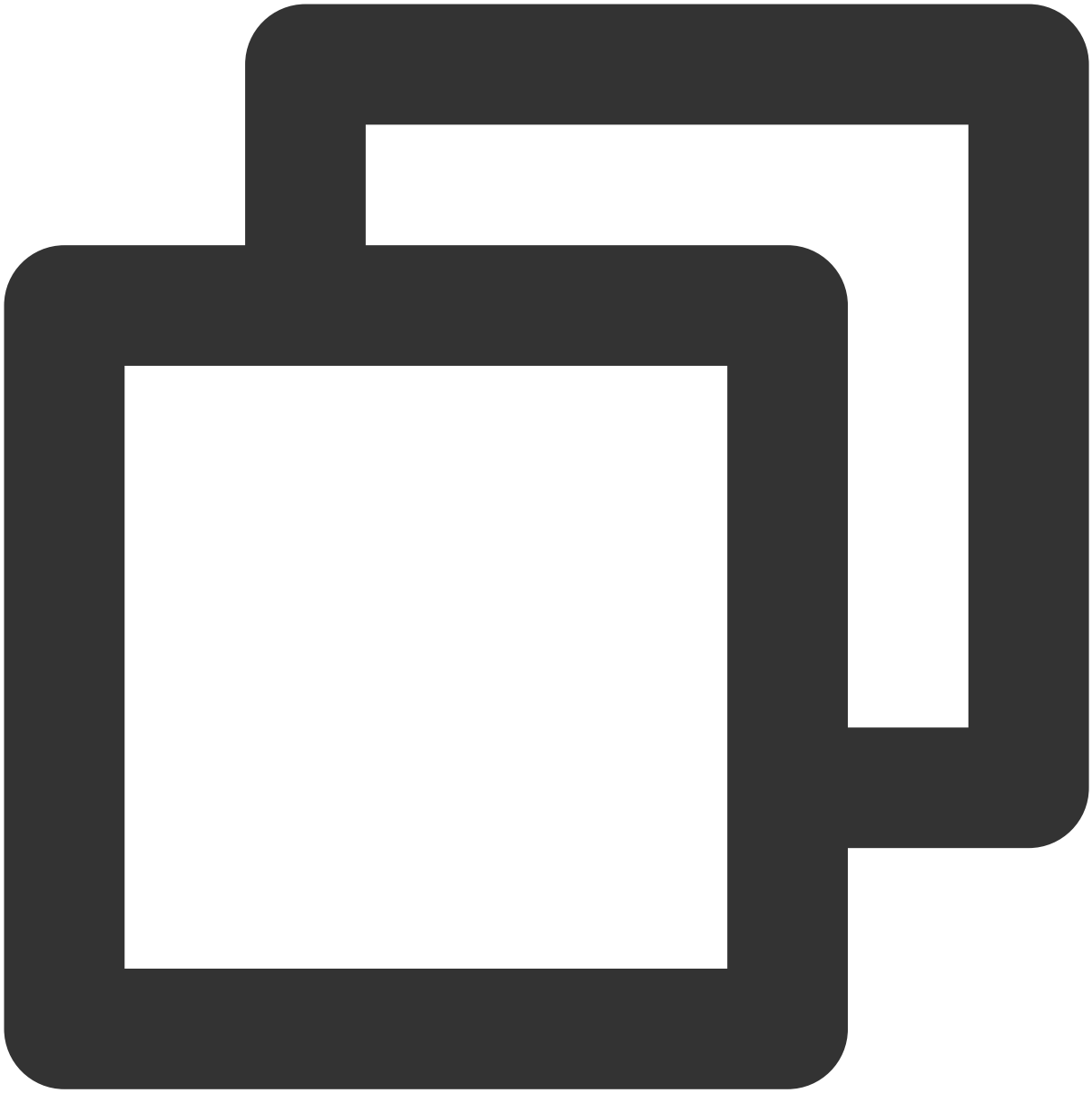


```
{
  "@timestamp": 1648803500.63659,
  "@filepath": "/var/log/tke-log-agent/test7/c816991f-adfe-4617-8cf3-9997aea90ded/c
  "log": "15:00:00.000[4349811564226374227] [http-nio-8081-exec-64] INFO  com.qclou
```

```
  "kubernetes": {
    "pod_name": "tke-es-687995d557-n29jr",
    "namespace_name": "default",
    "pod_id": "c816991f-adfe-4617-8cf3-9997aea90ded",
    "labels": {
      "k8s-app": "tke-es",
      "pod-template-hash": "687995d557",
      "qcloud-app": "tke-es"
    },
    "annotations": {
      "qcloud-redeploy-timestamp": "1648016531476",
      "tke.cloud.tencent.com/networks-status": "[{\\n    \\"name\\": \\"tke-bridge\
    },
    "host": "10.0.96.47",
    "container_name": "nginx",
    "docker_id": "add90ccf49626ef42d5615a636aae74d6380996043cf6f6560d8131f21a4d8ba"
    "container_hash": "nginx@sha256:e1211ac17b29b585ed1aee166a17fad63d344bc973bc638
    "container_image": "nginx"
  }
}
```


当日志采集后投递进 ElasticSearch 时，由于支持嵌套 JSON 结构，因此不需要对数据做出太大改动。但当需要投递到数据库时，此时就需要将数据转换成能够识别的单层 JSON 格式，因此按照下文顺序设计解析流程：

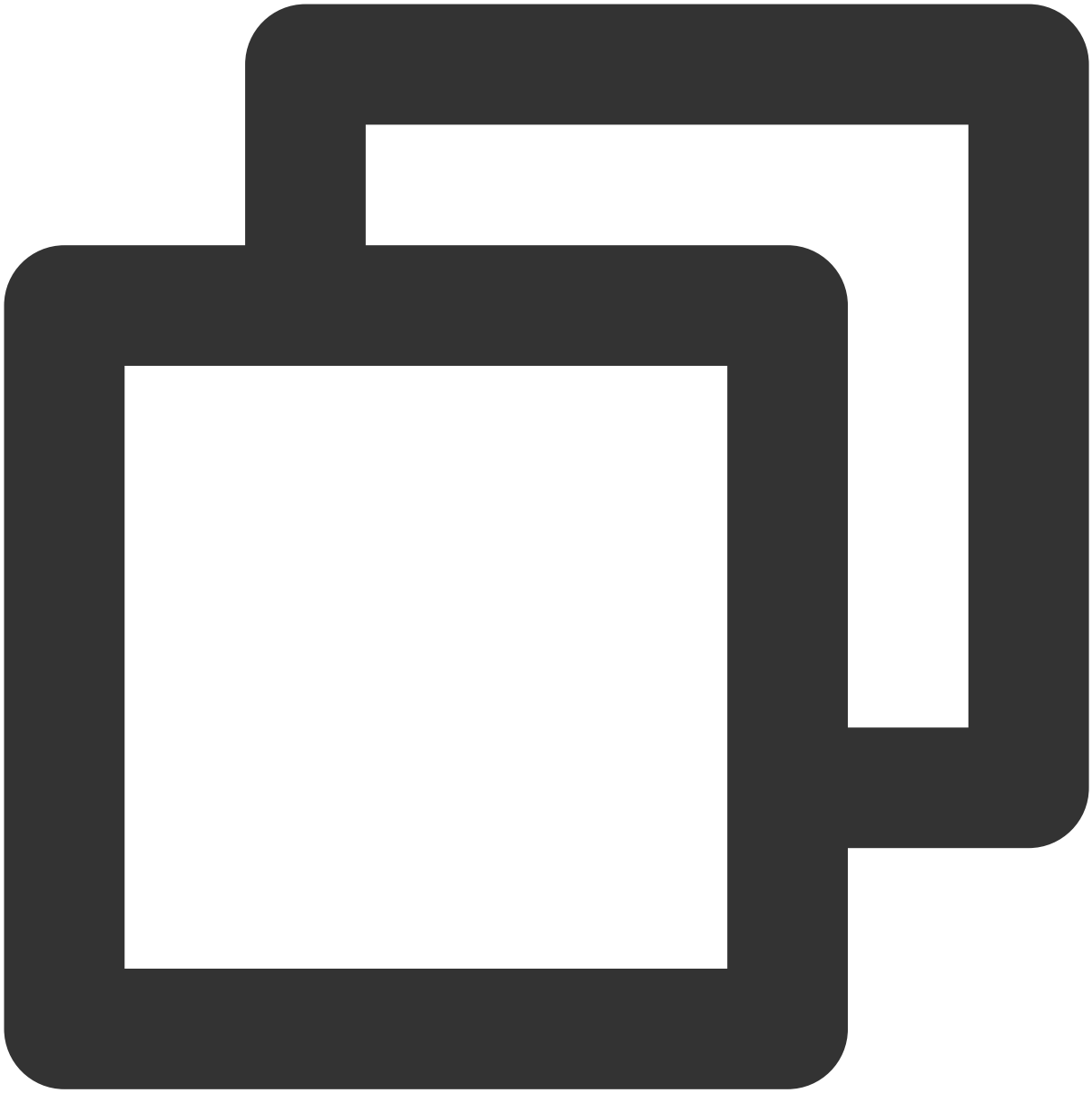
1. 使用 **JSONPATH** 语句处理第一层嵌套的 JSON 结构 `$.kubernetes` ，解析模式选择 `JSON` 。首先将嵌套 JSON 结构转换为单层 JSON 结构。测试后结果如下所示：




```
{
  "@timestamp": 1.64880350063659E9,
  "@filepath": "/var/log/tke-log-agent/test7/c816991f-adfe-4617-8cf3-9997aea90ded/c
  "log": "15:00:00.000[4349811564226374227] [http-nio-8081-exec-64] INFO  com.qclou
  "$.kubernetes.pod_name": "tke-es-687995d557-n29jr",
  "$.kubernetes.namespace_name": "default",
  "$.kubernetes.pod_id": "c816991f-adfe-4617-8cf3-9997aea90ded",
  "$.kubernetes.labels": {
    "k8s-app": "tke-es",
    "pod-template-hash": "687995d557",
    "qcloud-app": "tke-es"
```

```
  },
  "$.kubernetes.annotations": {
    "qcloud-redeploy-timestamp": "1648016531476",
    "tke.cloud.tencent.com/networks-status": "[{\\n    \\"name\\": \\"tke-bridge\\"
  },
  "$.kubernetes.host": "10.0.96.47",
  "$.kubernetes.container_name": "nginx",
  "$.kubernetes.docker_id": "add90ccf49626ef42d5615a636aae74d6380996043cf6f6560d813
  "$.kubernetes.container_hash": "nginx@sha256:e1211ac17b29b585ed1aee166a17fad63d34
  "$.kubernetes.container_image": "nginx"
}
```

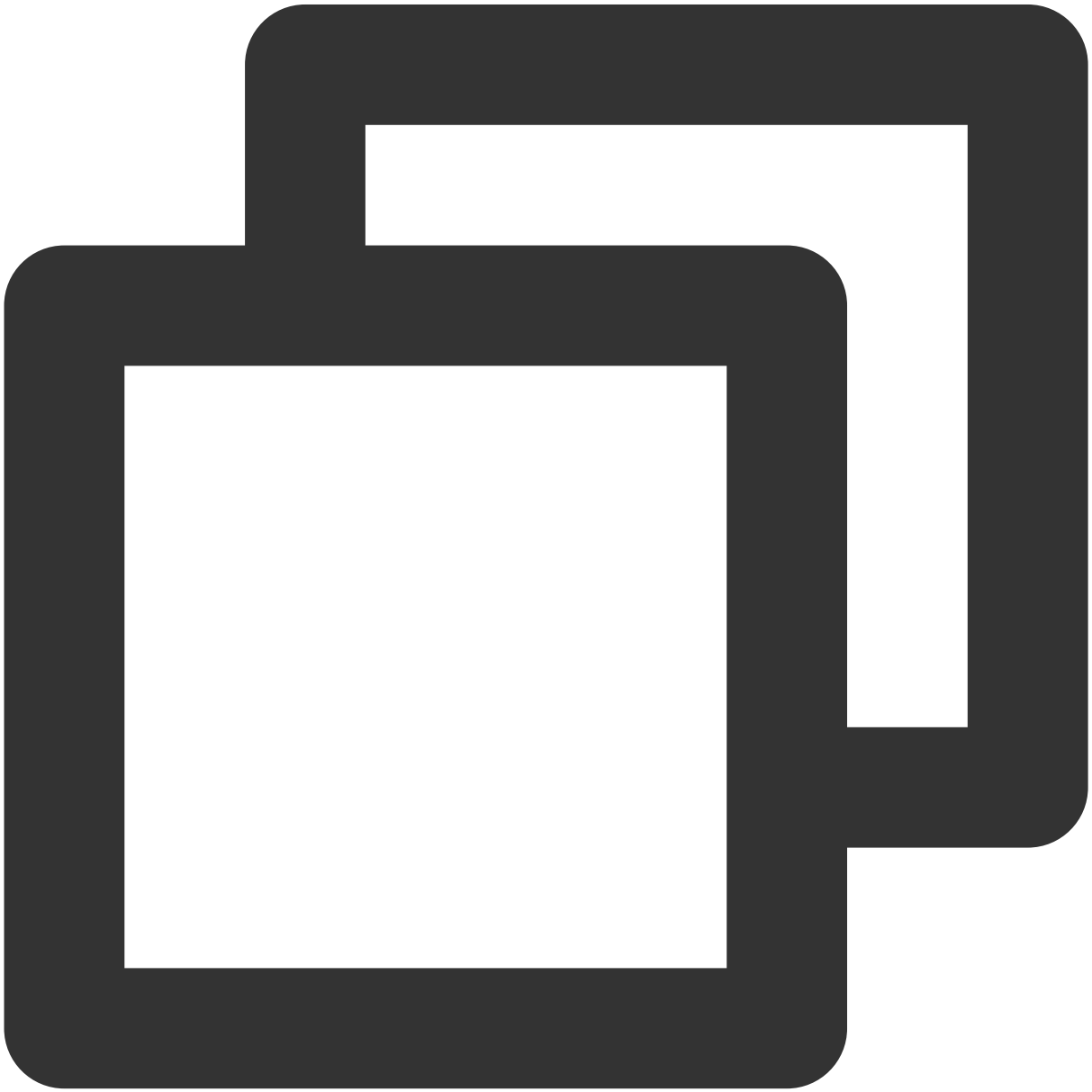
2. 依次处理第二层嵌套结构 `$.kubernetes.annotations` 和 `$.kubernetes.labels` 。在处理链中使用 `Map` 方式选中这两个名称，即可将嵌套格式转换成单层 JSON 格式。处理后如下所示：




```
{
  "@timestamp": 1648803500.63659,
  "@filepath": "/var/log/tke-log-agent/test7/c816991f-adfe-4617-8cf3-9997aea90ded/c
  "log": "15:00:00.000[4349811564226374227] [http-nio-8081-exec-64] INFO  com.qclou
  "$.kubernetes.pod_name": "tke-es-687995d557-n29jr",
  "$.kubernetes.namespace_name": "default",
  "$.kubernetes.pod_id": "c816991f-adfe-4617-8cf3-9997aea90ded",
  "$.kubernetes.host": "10.0.96.47",
  "$.kubernetes.container_name": "nginx",
  "$.kubernetes.docker_id": "add90ccf49626ef42d5615a636aae74d6380996043cf6f6560d813
  "$.kubernetes.container_hash": "nginx@sha256:e1211ac17b29b585ed1aee166a17fad63d34
```

```
    "$.kubernetes.container_image": "nginx",
    "$.kubernetes.labels.k8s-app": "tke-es",
    "$.kubernetes.labels.pod-template-hash": "687995d557",
    "$.kubernetes.labels.qcloud-app": "tke-es",
    "$.kubernetes.annotations.qcloud-redeploy-timestamp": "1648016531476",
    "$.kubernetes.annotations.tke.cloud.tencent.com/networks-status": "[{\\n    \\"na
}
```

3. 最后修改相应映射字段的 `key` 为所需名称，并且删去不需要的字段。此时单击**添加处理链**后，打开**处理上层所有结果**的按钮，整理优化后可以参考如下所示：



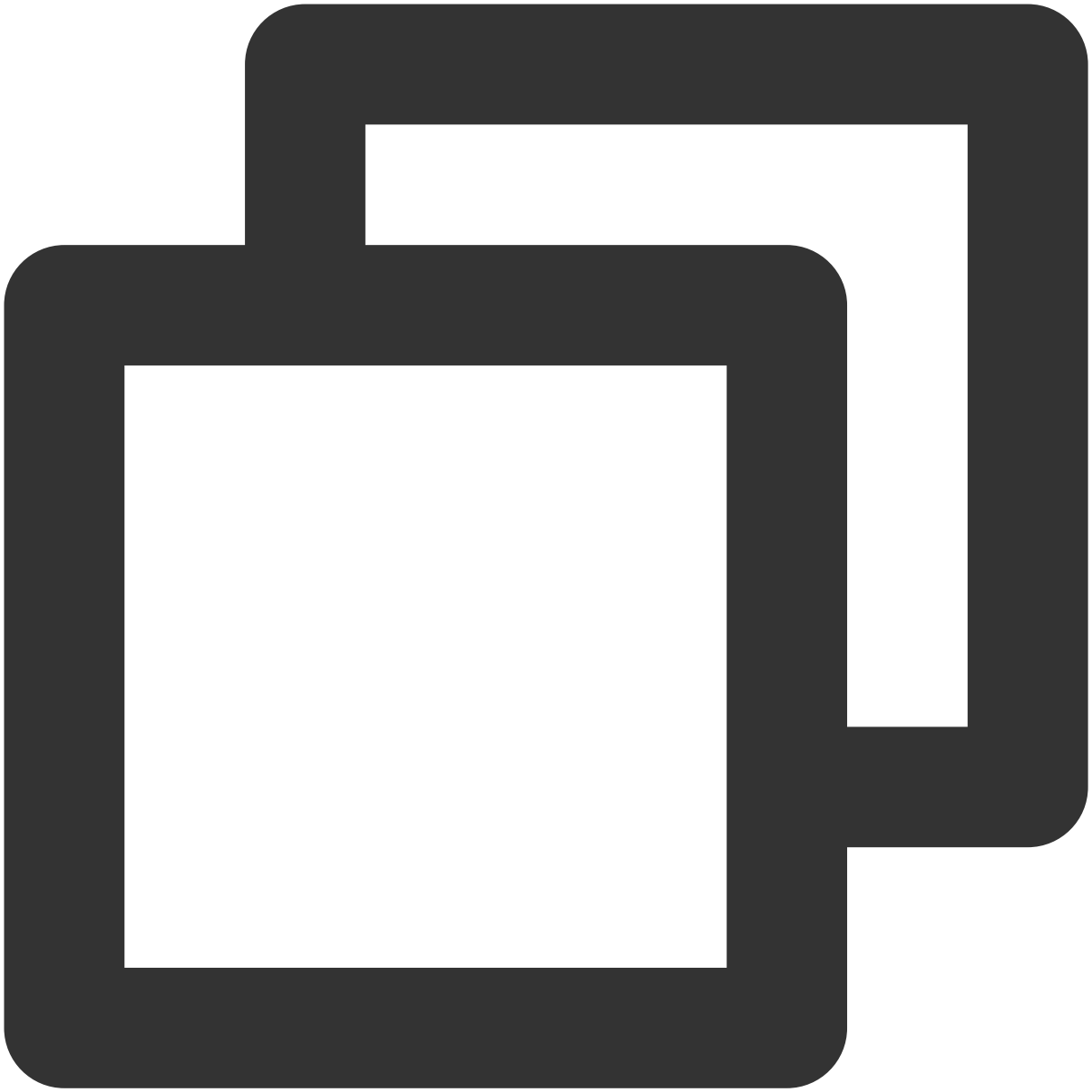
```
{
  "@timestamp": 1.64880350063659E9,
  "@filepath": "/var/log/tke-log-agent/test7/c816991f-adfe-4617-8cf3-9997aea90ded/c
  "log": "15:00:00.000[4349811564226374227] [http-nio-8081-exec-64] INFO  com.qclou
  "pod_name": "tke-es-687995d557-n29jr",
  "namespace_name": "default",
  "pod_id": "c816991f-adfe-4617-8cf3-9997aea90ded",
  "host": "10.0.96.47",
  "container_name": "nginx",
  "docker_id": "add90ccf49626ef42d5615a636aae74d6380996043cf6f6560d8131f21a4d8ba"
}
```

说明：

在使用 **JSONPath** 处理参数时，当 `key` 中本身带有英文句号 `.` 时，需要在路径中方括号和单引号进行隔离。例如 `{"key1.key2":"value1"}` 中，要想取得对应字段，则需要使用 `$.['key1.key2']` 进行获取相应键值。

处理字符串序列化 JSON 格式日志

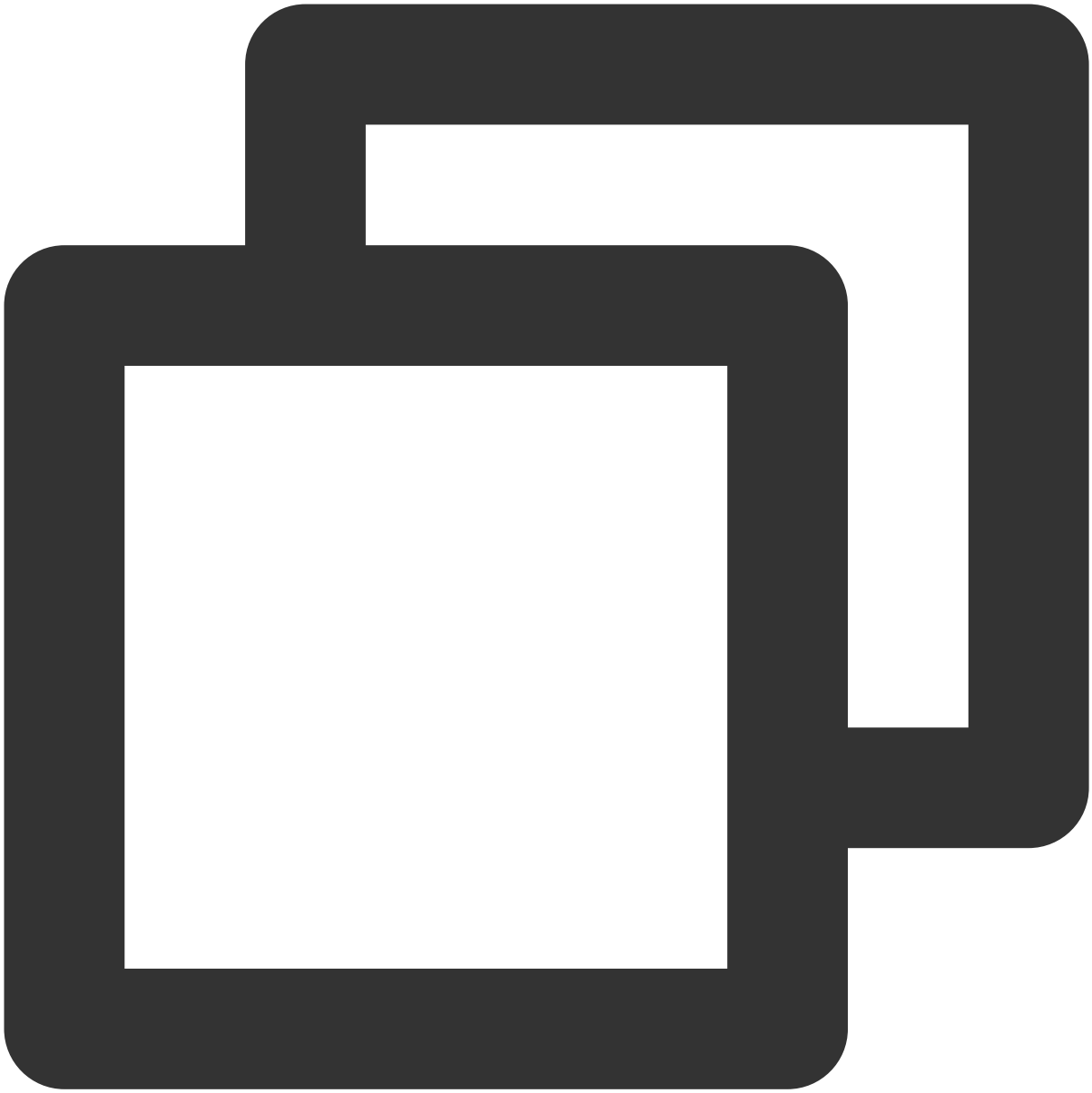
有时 JSON 格式在传输过程中，由于格式或性能等需要，需要转义成字符串格式的形式，此处把这种格式叫做 **Raw JSON**。需要在数据处理中重新将字符串反序列化成 JSON 格式。 此处以 MongoDB 中的 Raw JSON 格式为例，大致结构如下所示：



```
{
  "key": "  {\\n    \\"categories\\": [\\"dev\\"],\\n    \\"created_at\\": \\"2020-
}
```

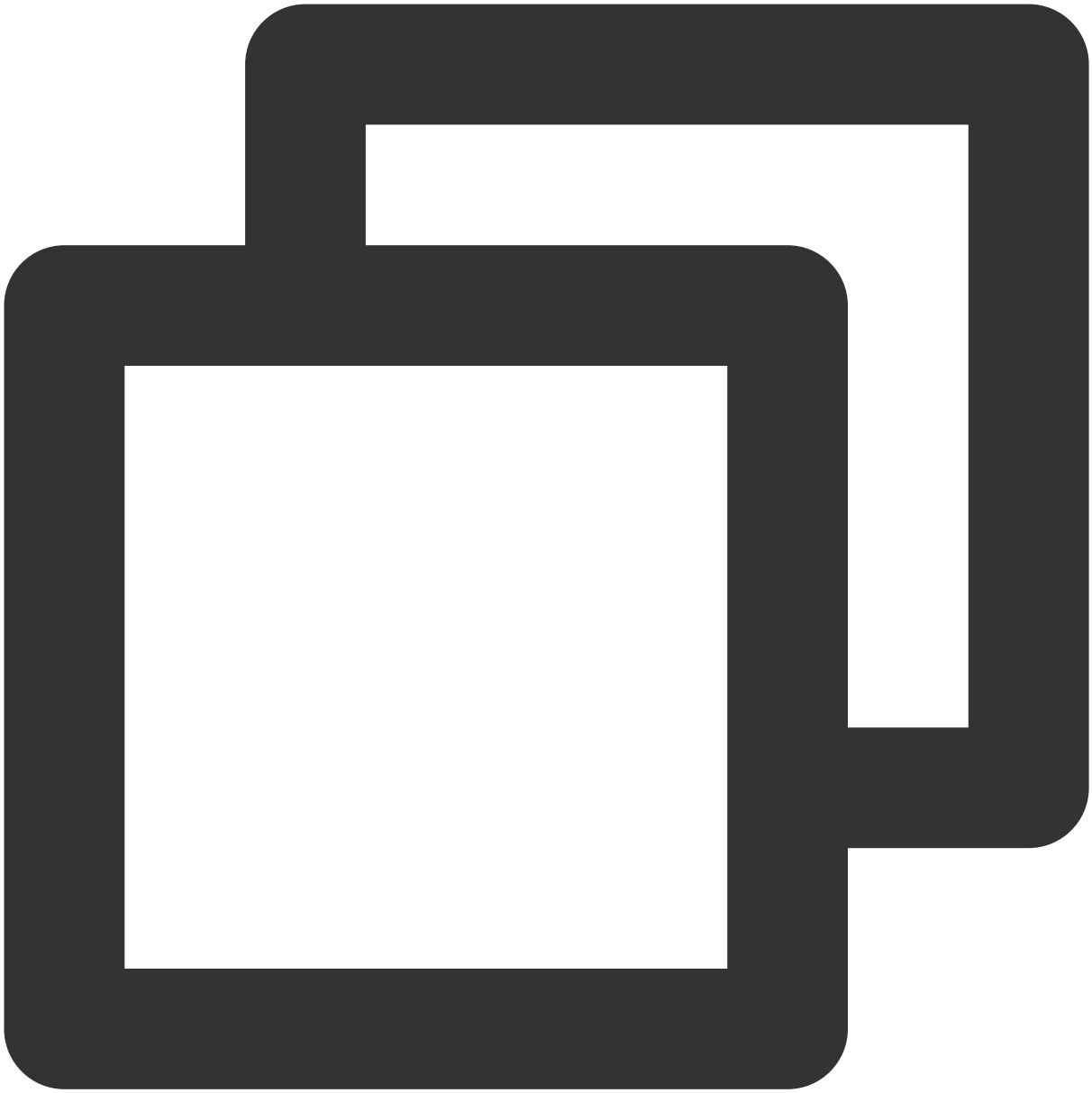
如果在数据处理中就将 Raw JSON 转换成普通的 JSON 格式，就能按照字段格式直接投递到目标下游中。大致处理思路如下所示：

1. 首先设置 **解析模式** 为 JSON。这样能够将消息先预读取成内部的 MAP 格式。解析后如下所示：



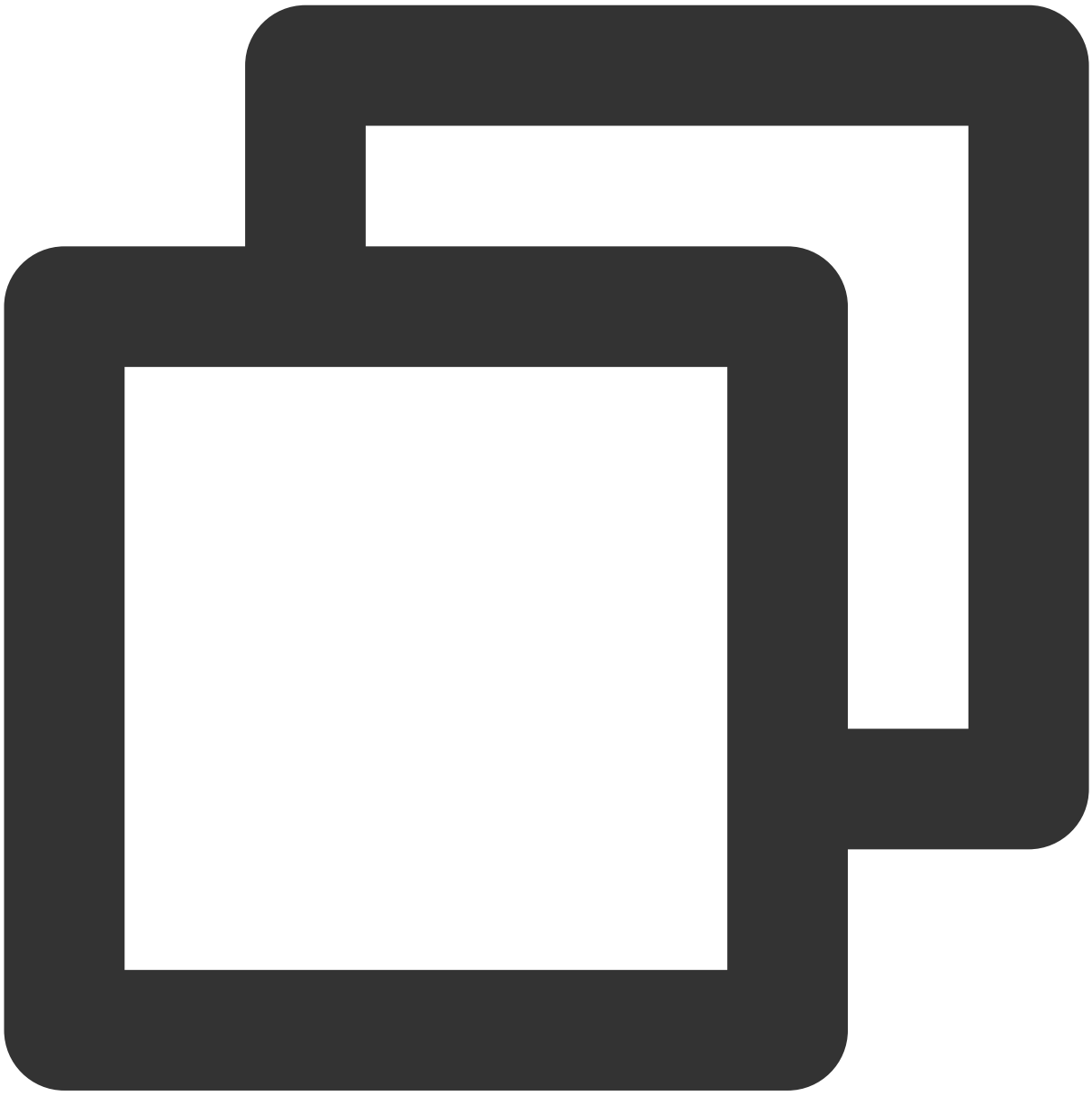
```
{
  "key": "  {\\n    \\"categories\\": [\\"dev\\"],\\n    \\"created_at\\": \\"2020-
}
```

2. 单击 **添加处理链**，使用 MAP 方式选中 key，解析方式选择 `JSON` 。这样数据处理就能将 RAW JSON 自动转换成 JSON 形式。解析后如下所示：




```
{
  "key.categories": [
    "dev"
  ],
  "key.created_at": "2020-01-05 13:42:19.324003",
  "key.icon_url": "https://assets.chucknorris.host/img/avatar/chuck-norris.png",
  "key.id": "elgv2wkvt8ioag6xywykbq",
  "key.updated_at": "2020-01-05 13:42:19.324003",
  "key.url": "https://api.chucknorris.io/jokes/elgv2wkvt8ioag6xywykbq",
  "key.value": "Chuck Norris's keyboard doesn't have a Ctrl key because nothing con
}
```

3. 最后单击 **添加处理链** 后，打开**处理上层所有结果**的按钮，修改相应映射字段的 `key` 为所需名称，并且删去不需要的字段。整理优化后可以参考如下所示：



```
{
  "categories": [
    "dev"
  ],
  "created_at": "2020-01-05 13:42:19.324003",
  "icon_url": "https://assets.chucknorris.host/img/avatar/chuck-norris.png",
```

```
    "id": "elgv2wkvt8ioag6xywykbq",
    "updated_at": "2020-01-05 13:42:19.324003",
    "url": "https://api.chucknorris.io/jokes/elgv2wkvt8ioag6xywykbq",
    "value": "Chuck Norris's keyboard doesn't have a Ctrl key because nothing control
  }
```

说明：

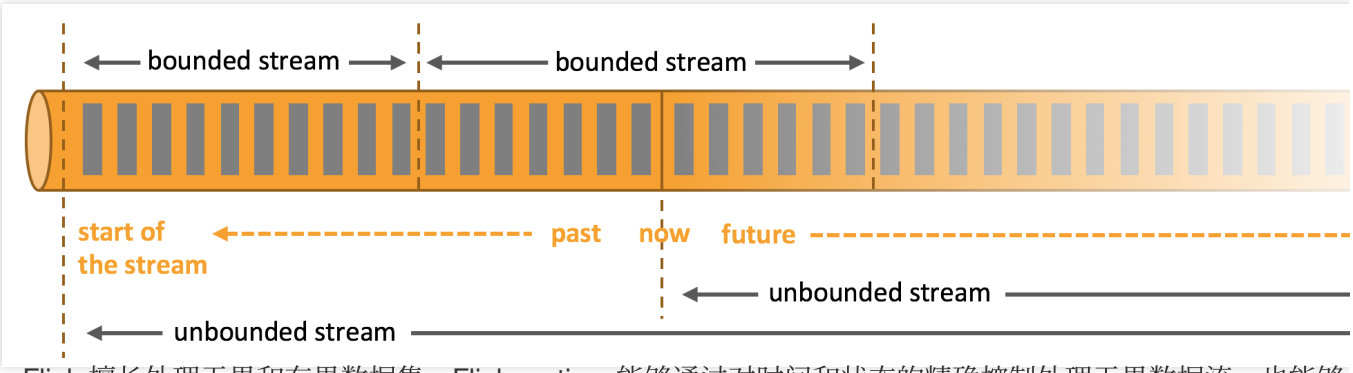
目前 RAW JSON 只支持解析 MAP 类型的数据。当最外层为 List 类型时，例如 `"[\\"test1\\",\\"test2\\"]"`，或者 `"[{\\"key\\":\\"value\\"}]"`，由于无法解析合适的键值，因此将会提示解析失败。

Flink 接入 CKafka

最近更新时间：2024-05-31 12:08:45

Flink 简介

Apache Flink 是一个可以处理流数据的实时处理框架，用于在无界和有界数据流上进行有状态的计算。Flink 能在所有常见集群环境中运行，并能以内存速度和任意规模进行计算。



Apache Flink 擅长处理无界和有界数据集。Flink runtime 能够通过对时间和状态的精确控制处理无界数据流，也能够使用为固定大小数据集设计的算法和数据结构对有界数据集进行处理，并达到出色的性能。
应用程序可能会使用来自各种数据源（如消息队列或分布式日志，如 Apache Kafka 或 Kinesis）的实时数据。Flink 提供了 Apache Kafka 连接器，用于从 Kafka topic 中读取或者向其中写入数据，可提供一次精确的处理语义。

操作步骤

步骤1：获取 CKafka 实例接入地址

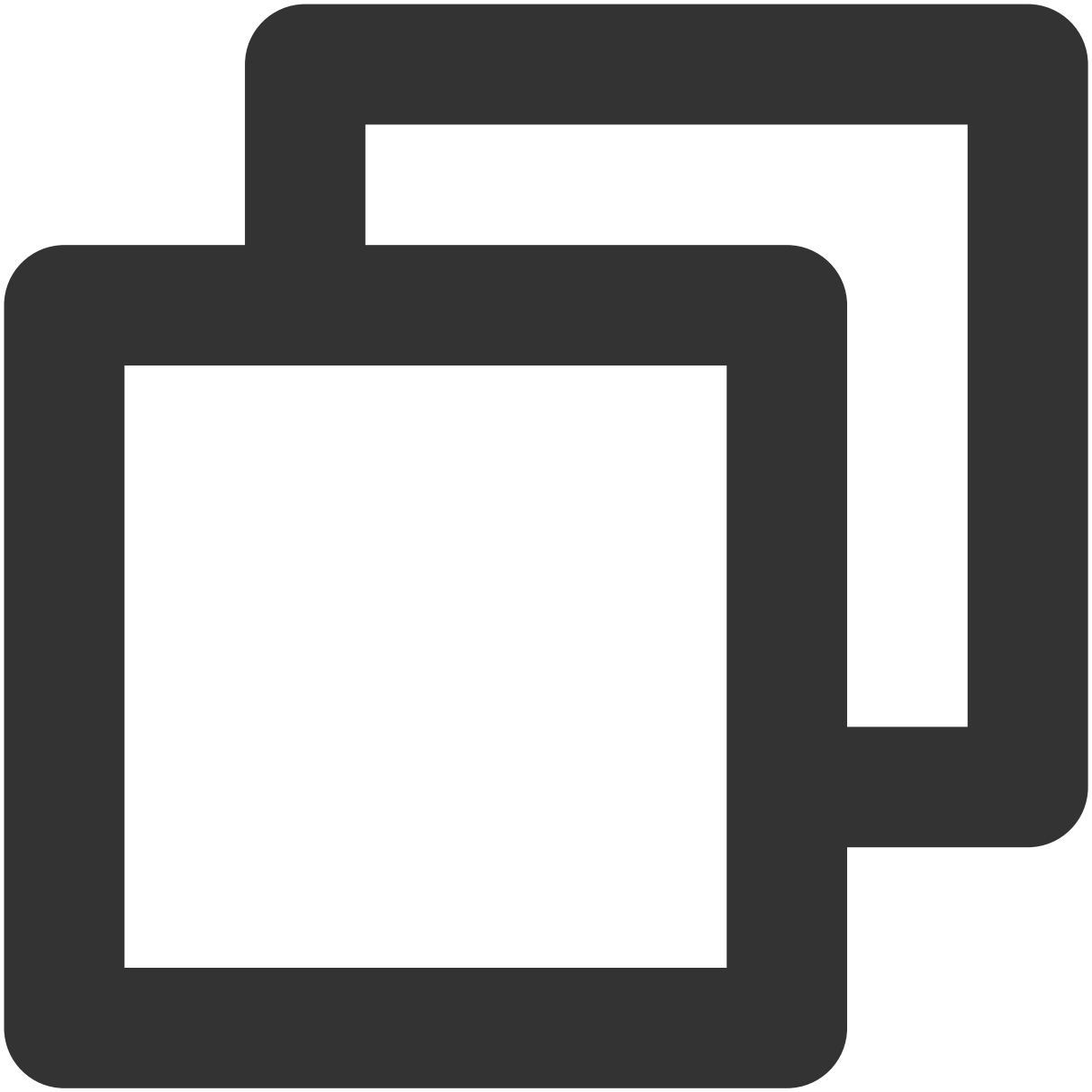
1. 登录 CKafka 控制台。
2. 在左侧导航栏选择**实例列表**，单击实例的“ID”，进入实例基本信息页面。
3. 在实例的基本信息页面的**接入方式**模块，可获取实例的接入地址，接入地址是生产消费需要用到的 bootstrap-server。

步骤2：创建 Topic

1. 在实例基本信息页面，选择顶部**Topic管理**页签。
2. 在 Topic 管理页面，单击**新建**，创建一个名为 test 的 Topic，接下来将以该 Topic 为例介绍如何消费。

步骤3：添加 Maven 依赖

pom.xml 配置如下：



```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.example</groupId>
    <artifactId>Test-CKafka</artifactId>
    <version>1.0-SNAPSHOT</version>
    <dependencies>
        <dependency>
```

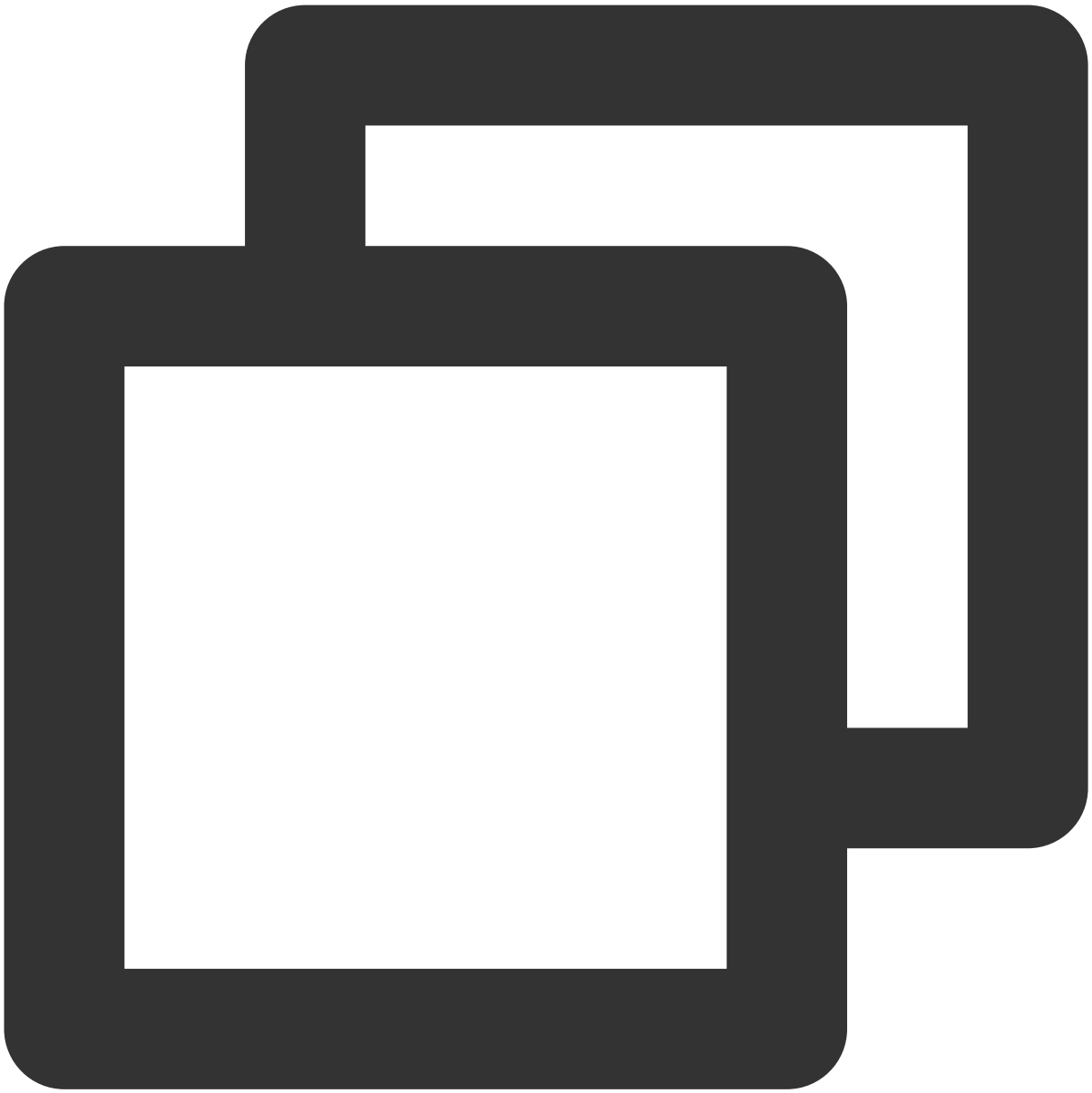
```
            <groupId>org.apache.kafka</groupId>
            <artifactId>kafka-clients</artifactId>
            <version>0.10.2.2</version>
        </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-simple</artifactId>
            <version>1.7.25</version>
            <scope>compile</scope>
        </dependency>
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-java</artifactId>
            <version>1.6.1</version>
        </dependency>
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-streaming-java_2.11</artifactId>
            <version>1.6.1</version>
        </dependency>
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-connector-kafka_2.11</artifactId>
            <version>1.7.0</version>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.3</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

步骤4：消费 CKafka 中的消息

您可以单击以下页面，查看消费消息的两种方式。通过控制台或打印的日志即可查看消费结果。

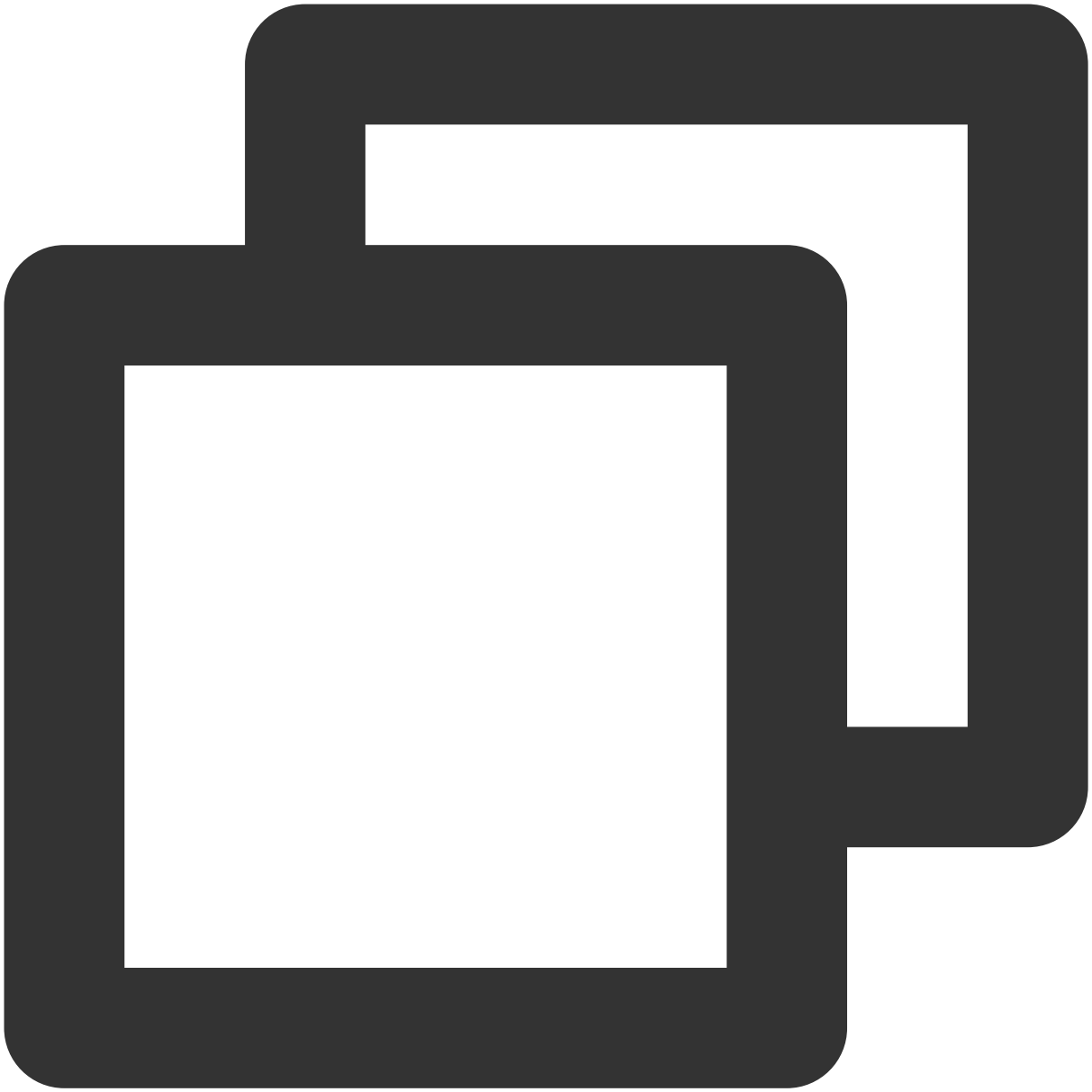
通过 VPC 方式消费

通过公网域名方式消费



```
import org.apache.flink.api.common.serialization.SimpleStringSchema;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer;
import java.util.Properties;

public class CKafkaConsumerDemo {
        public static void main(String args[]) throws Exception {
                StreamExecutionEnvironment env = StreamExecutionEnvironment.getExec
```

```
                Properties properties = new Properties();
                //公网接入域名地址,即公网路由地址,在实例详情页的接入方式模块获取。
                properties.setProperty("bootstrap.servers", "IP:PORT");
                //消费者组id。
                properties.setProperty("group.id", "testConsumerGroup");
                DataStream<String> stream = env
                                .addSource(new FlinkKafkaConsumer<>("topicName", ne
                stream.print();
                env.execute();
        }
}
```



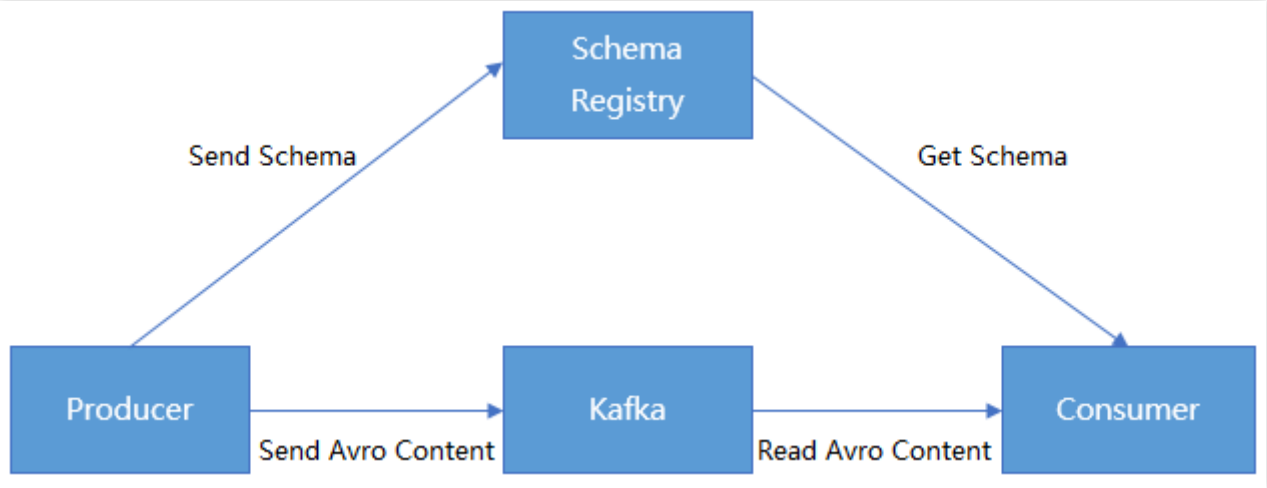
```
import org.apache.flink.api.common.serialization.SimpleStringSchema;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer;
import java.util.Properties;

public class CKafkaConsumerDemo {
        public static void main(String args[]) throws Exception {
                StreamExecutionEnvironment env = StreamExecutionEnvironment.getExec
                Properties properties = new Properties();
                //公网接入域名地址,即公网路由地址,在实例详情页的接入方式模块获取。
```

```
                properties.setProperty("bootstrap.servers", "IP:PORT");
                //消费者组id。
                properties.setProperty("group.id", "testConsumerGroup");
                properties.setProperty("security.protocol", "SASL_PLAINTEXT");
                properties.setProperty("sasl.mechanism", "PLAIN");
                //用户名和密码，注：用户名是需要拼接，并非控制台的用户名：instanceId#username
                properties.setProperty("sasl.jaas.config",
                                                        "org.apache.kafka.comm
                properties.setProperty("sasl.kerberos.service.name","kafka");
                DataStream<String> stream = env
                                .addSource(new FlinkKafkaConsumer<>("topicName", ne
                stream.print();
                env.execute();
        }
}
```

Schema Registry 接入 CKafka

最近更新时间：2024-07-19 14:25:57

无论是使用传统的 Avro API 自定义序列化类与反序列化类，还是使用 Twitter 的 Bijection 类库实现 Avro 的序列化与反序列化，两种方法有相同的缺点：在每条 Kafka 记录里都嵌入了 Schema，从而导致记录的大小成倍增加。但是不管怎样，在读取记录时仍然需要用到整个 Schema，所以要先找到 Schema。
CKafka 提供了数据共用一个 Schema 的方法：将 Schema 中的内容注册到 Confluent Schema Registry，Kafka Producer 和 Kafka Consumer 通过识别 Confluent Schema Registry 中的 schema 内容进行序列化和反序列化。



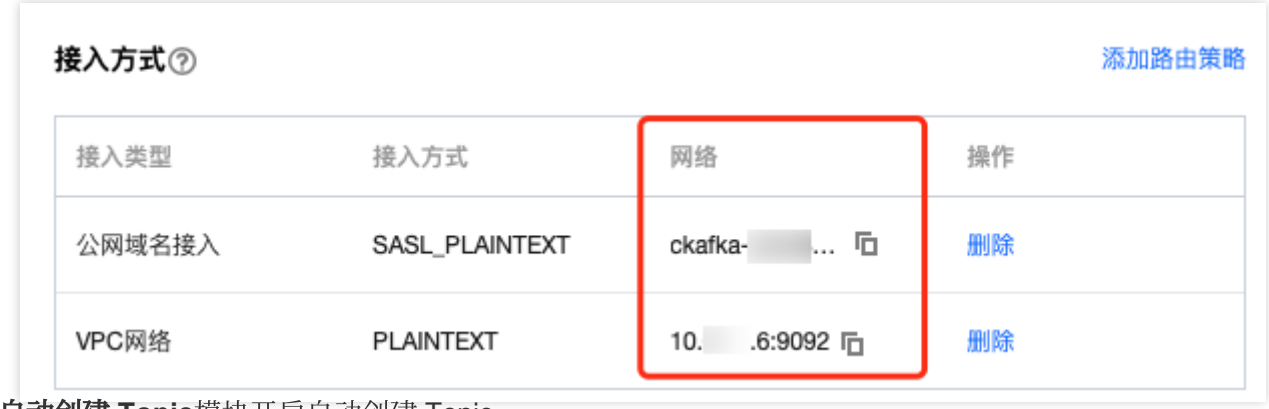
前提条件

下载 Download JDK 8。
下载 Confluent oss 4.1.1。
已 创建实例。

操作步骤

步骤1：获取实例接入地址并开启自动创建 Topic

1. 登录 CKafka 控制台。
2. 在左侧导航栏选择**实例列表**，单击实例的“ID”，进入实例基本信息页面。
3. 在实例的基本信息页面的**接入方式**模块，可获取实例的接入地址。



4. 在**自动创建 Topic**模块开启自动创建 Topic。

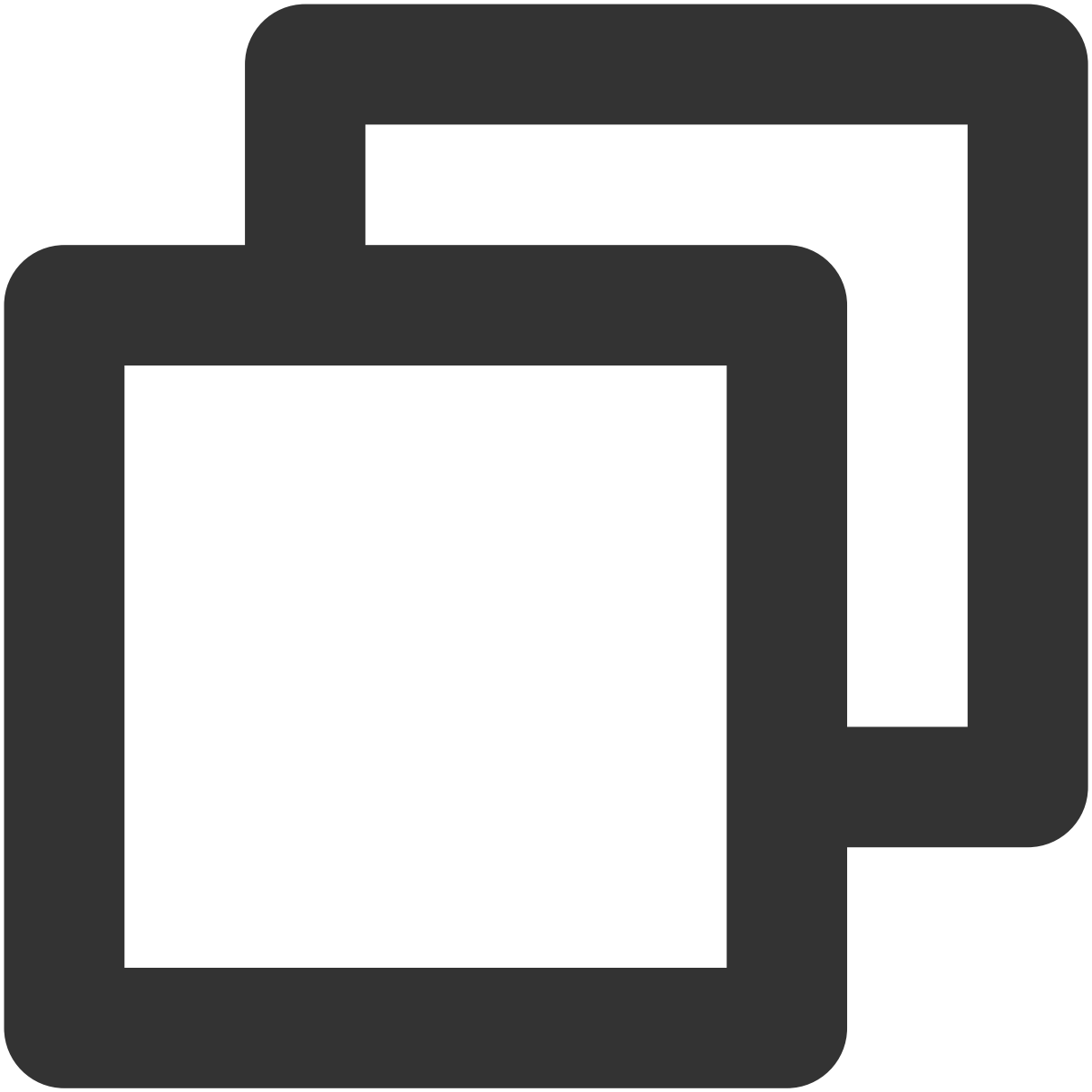
说明：

启动 oss 会创建 schemas 主题，所以实例中需要开启自动创建主题。

步骤2：准备 Confluent 配置

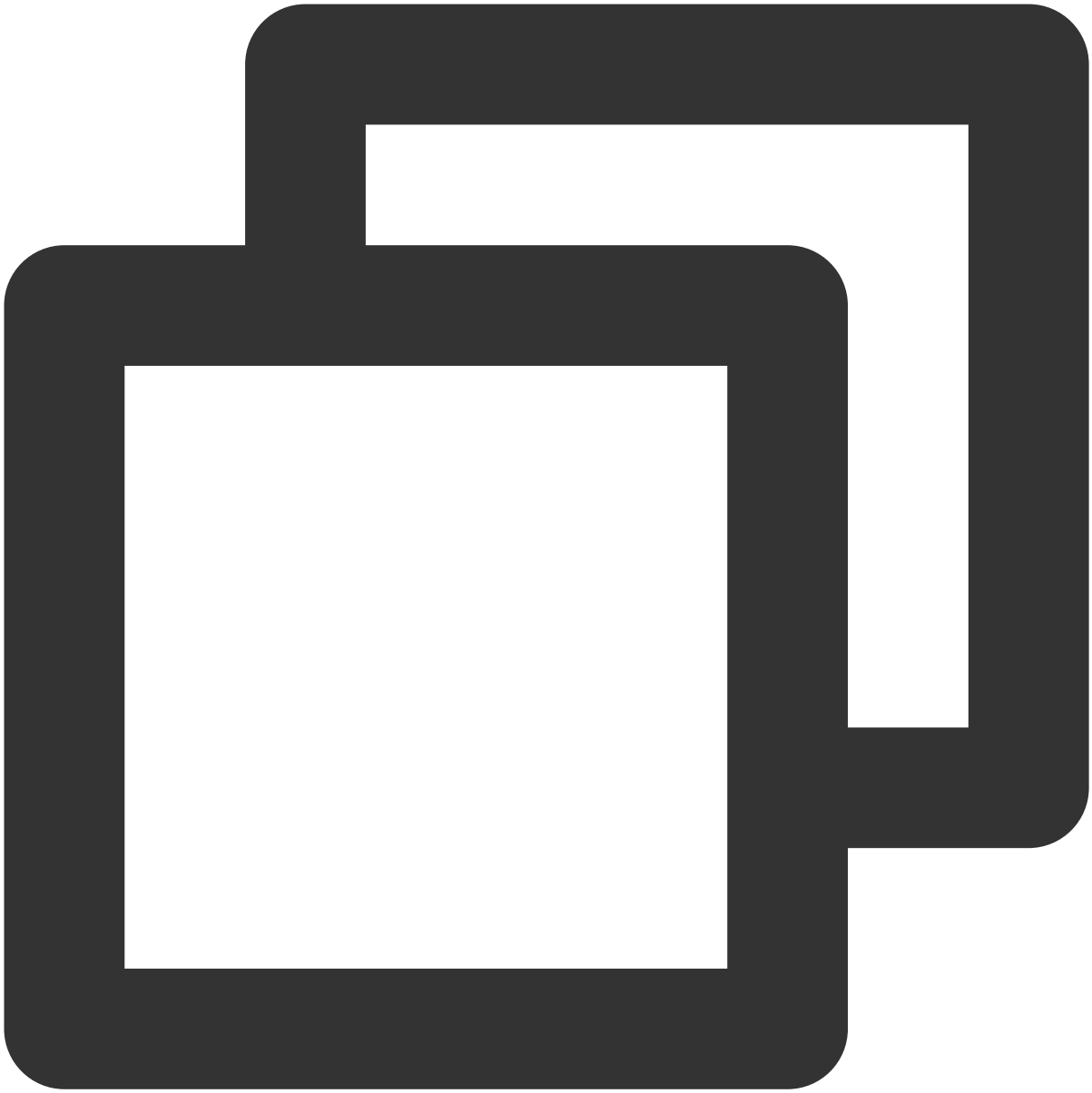
1. 修改 oss 配置文件中的 server 地址等信息。

PLAINTEXT 接入方式，配置信息如下：



```
kafkastore.bootstrap.servers=PLAINTEXT://xxxx
kafkastore.topic=schemas
debug=true
```

SASL_PLAINTEXT 接入方式，配置信息如下：



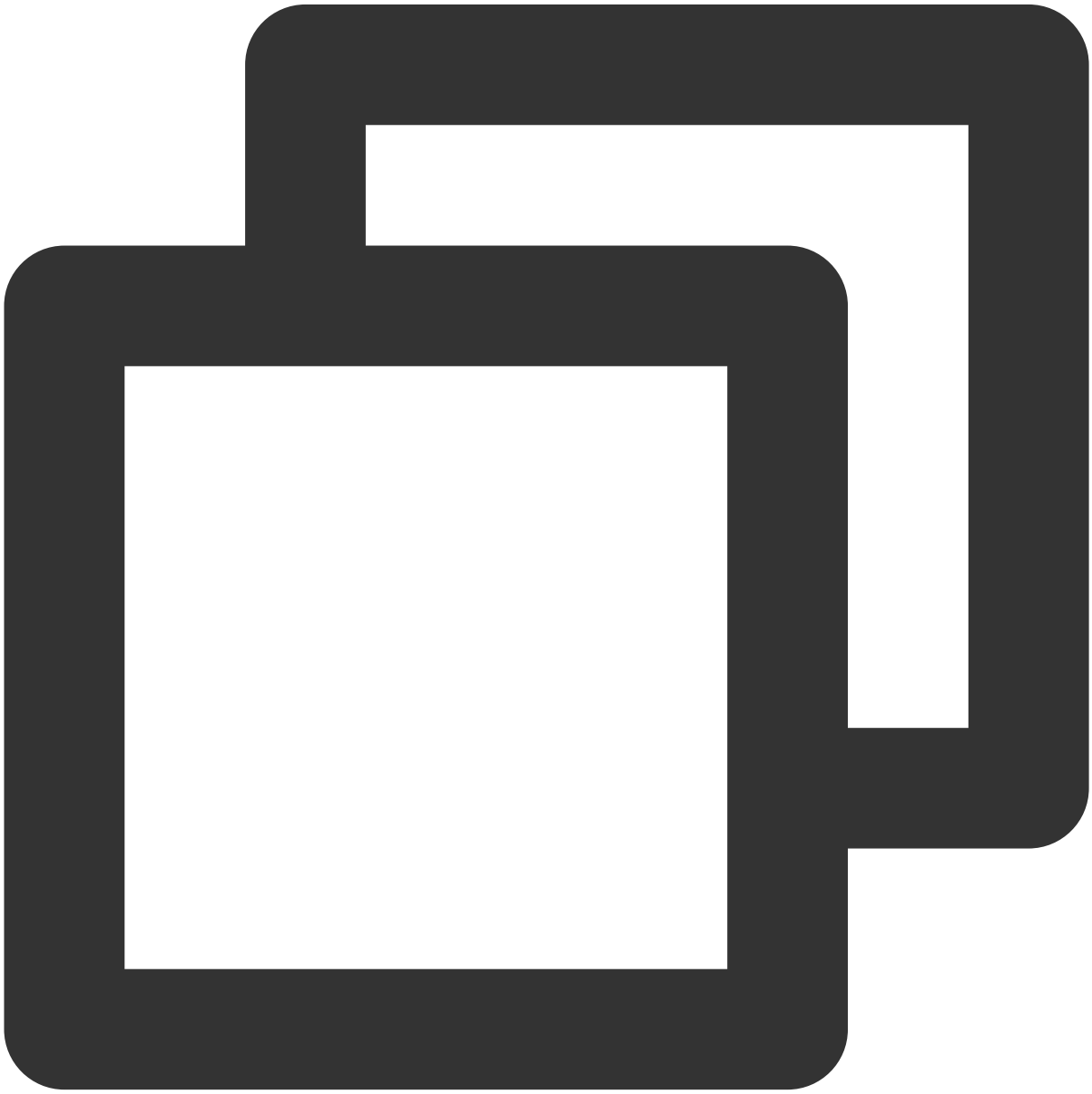
```
kafkastore.bootstrap.servers=SASL_PLAINTEXT://ckafka-xxxx.ap-xxx.ckafka.tencentclou
kafkastore.security.protocol=SASL_PLAINTEXT
kafkastore.sasl.mechanism=PLAIN
kafkastore.sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule
kafkastore.topic=schemas
debug=true
```

说明：

bootstrap.servers：接入网络，在 CKafka 控制台 的实例详情页面**接入方式**模块的网络列复制。

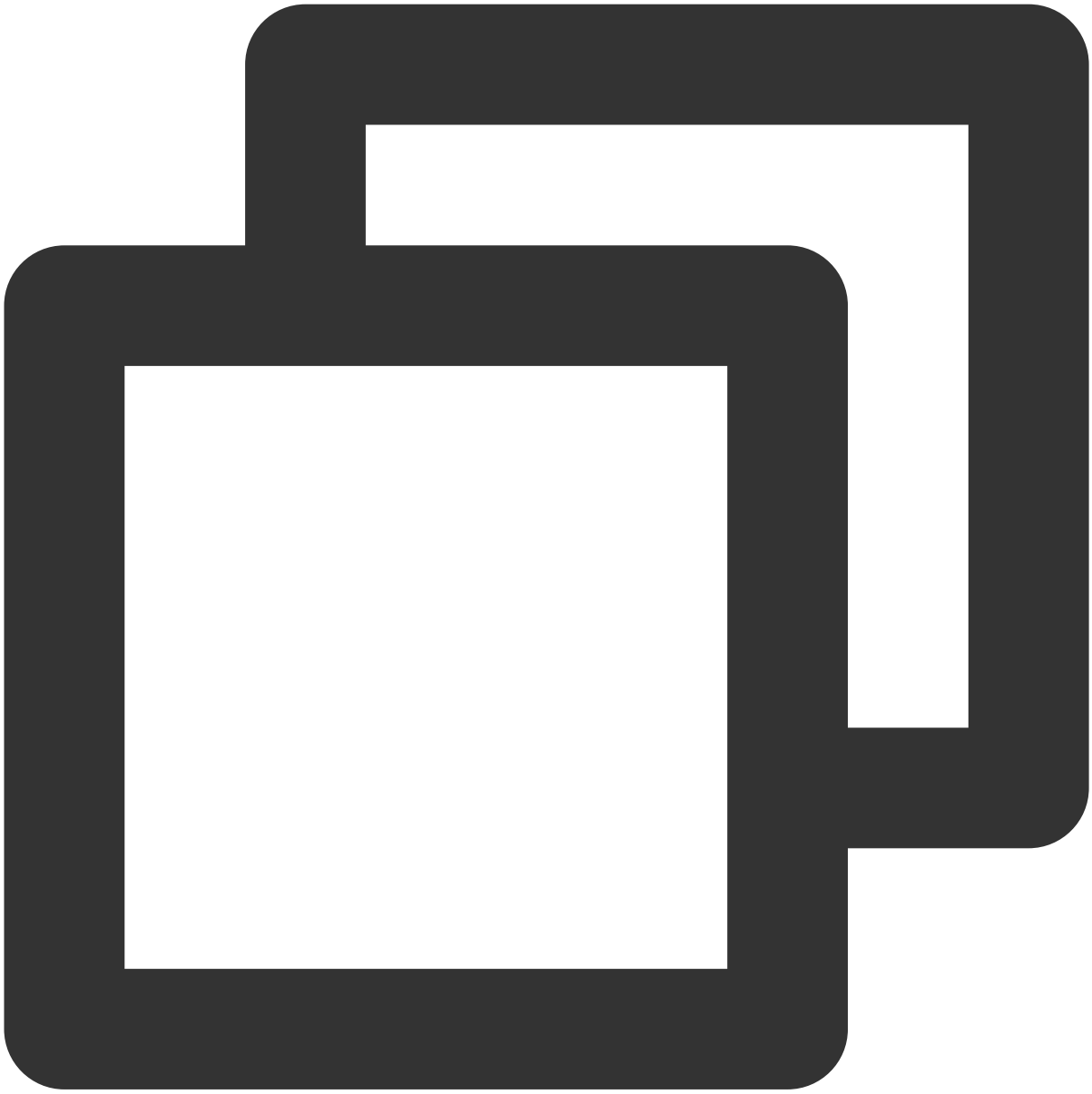
接入方式 添加路由策略

接入类型	接入方式	网络	操作
公网域名接入	SASL_PLAINTEXT	ckafka-...	删除
VPC网络	PLAINTEXT	172. .49:9	删除



```
<blockquote class="rno-document-tips rno-document-tips-explain">    <div class="rno
```

2. 执行如下命令启动 Schema Registry。



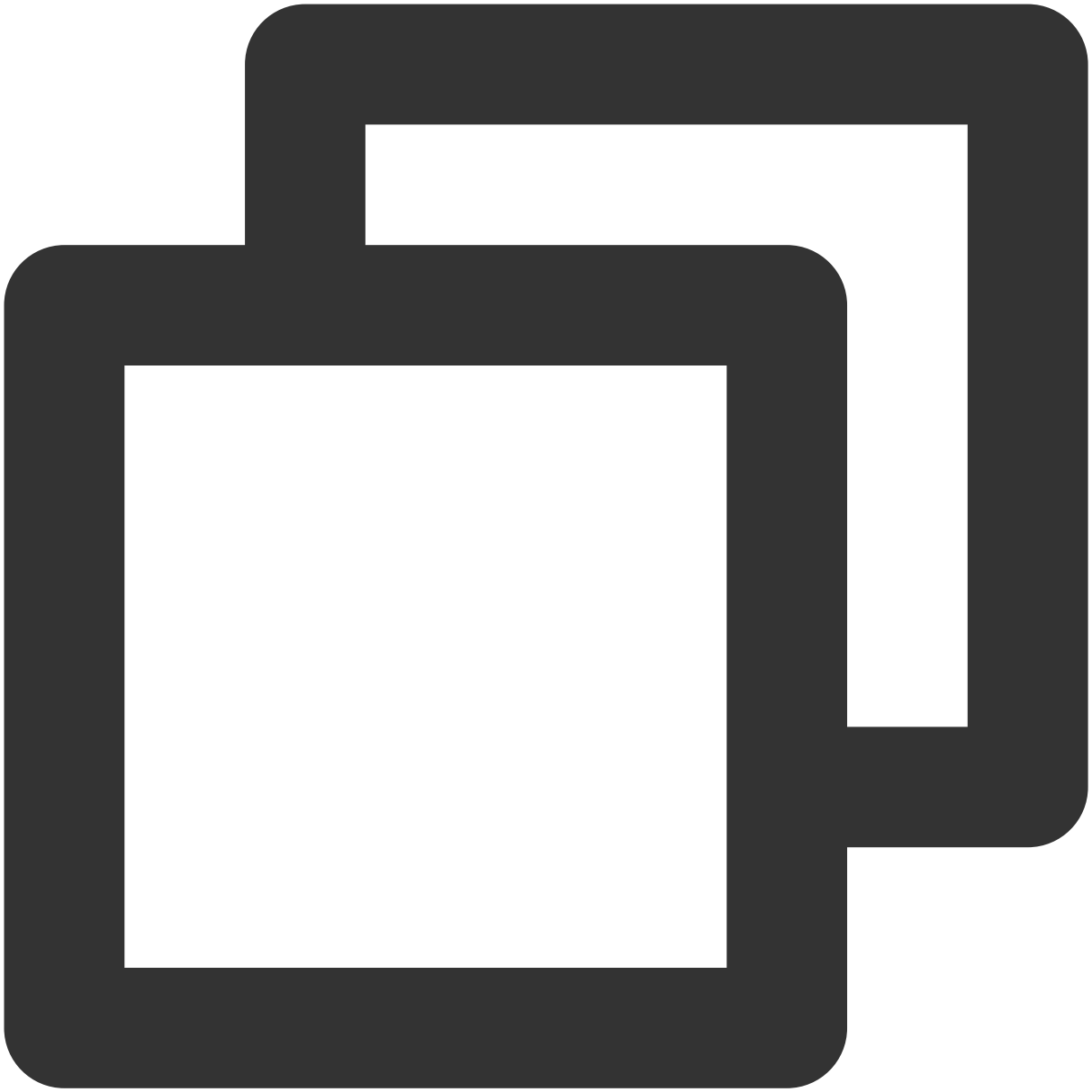
```
bin/schema-registry-start etc/schema-registry/schema-registry.properties
```

运行结果如下：

```
        kafkastore.init.timeout.ms = 60000
 (io.confluent.kafka.schemaregistry.rest.SchemaRegistryConfig:179)
[2019-07-09 16:33:24,889] INFO Logging initialized @523ms (org.eclipse.jetty.util.log:186)
[2019-07-09 16:33:25,607] INFO Initializing KafkaStore with broker endpoints: PLAINTEXT://172.
io.confluent.kafka.schemaregistry.storage.KafkaStore:103)
[2019-07-09 16:33:26,052] INFO Creating schemas topic _schemas (io.confluent.kafka.schemaregistr
afkaStore:186)
[2019-07-09 16:33:26,424] INFO Initialized last consumed offset to -1 (io.confluent.kafka.schema
orage.KafkaStoreReaderThread:138)
[2019-07-09 16:33:26,437] INFO [kafka-store-reader-thread-_schemas]: Starting (io.confluent.kafk
istry.storage.KafkaStoreReaderThread:66)
[2019-07-09 16:33:27,152] INFO Wait to catch up until the offset of the last message at 0 (io.co
ka.schemaregistry.storage.KafkaStore:277)
[2019-07-09 16:33:27,221] INFO Joining schema registry with Kafka-based coordination (io.conflue
hemaregistry.storage.KafkaSchemaRegistry:209)
[2019-07-09 16:33:27,316] INFO Finished rebalance with master election result: Assignment{versio
0, master='sr-1-/172.26.0.11-2019-07-09 16:33:27:278-f5aa186c-a6c2-4c93-95dd-[redacted]', mast
version=1,host=VM_0_11_centos,port=8081,scheme=http,masterEligibility=true} (io.confluent.kafka.
try.masterelector.kafka.KafkaGroupMasterElector:232)
[2019-07-09 16:33:27,336] INFO Wait to catch up until the offset of the last message at 1 (io.co
ka.schemaregistry.storage.KafkaStore:277)
[2019-07-09 16:33:27,458] INFO Adding listener: http://0.0.0.0:8081 (io.confluent.rest.Applicati
```

步骤3：收发消息

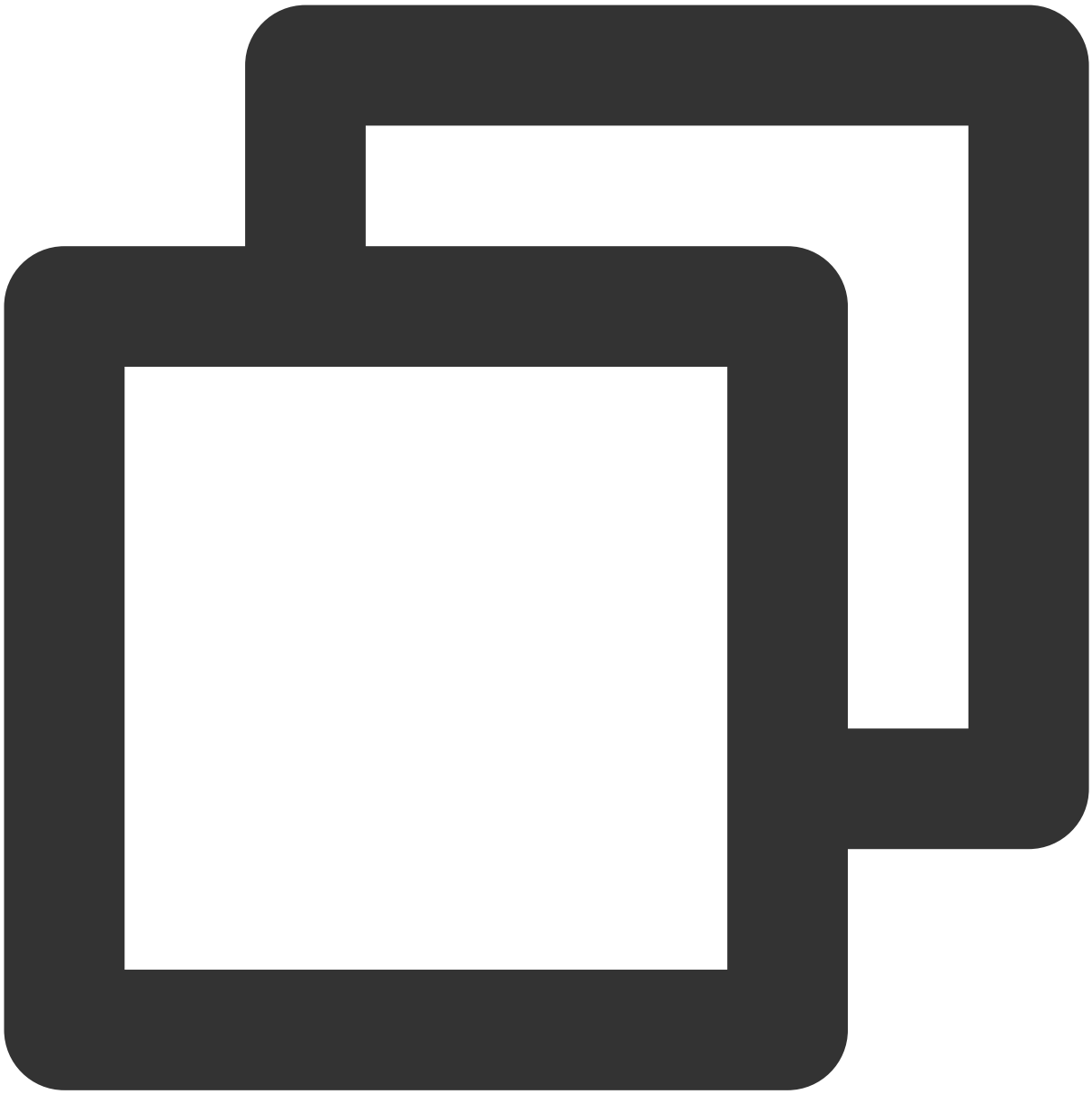
现有 schema 文件，其中内容如下：



```
{
    "type": "record",
    "name": "User",
    "fields": [
        {"name": "id", "type": "int"},
        {"name": "name",  "type": "string"},
        {"name": "age", "type": "int"}
    ]
}
```

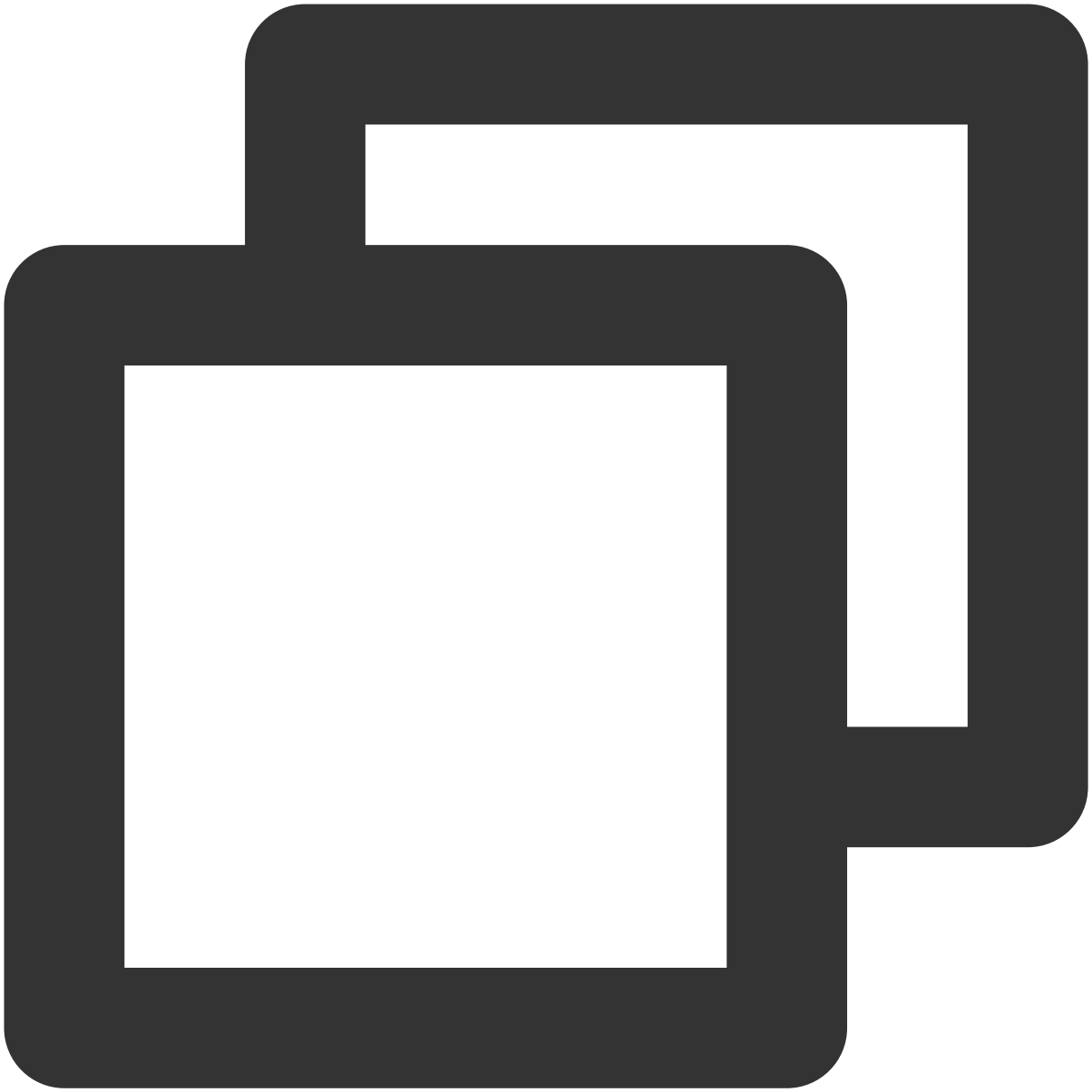
1. 注册 schema 到对应 Topic（注册 Topic 名为 test）。

下面的脚本是直接在 Schema Registry 部署的环境中使用 curl 命令调用对应 API 实现注册的一个示例：



```
curl -X POST -H "Content-Type: application/vnd.schemaregistry.v1+json" \\
--data '{"schema": "{\\"type\\": \\"record\\", \\"name\\": \\"User\\", \\"field
http://127.0.0.1:8081/subjects/test/versions
```

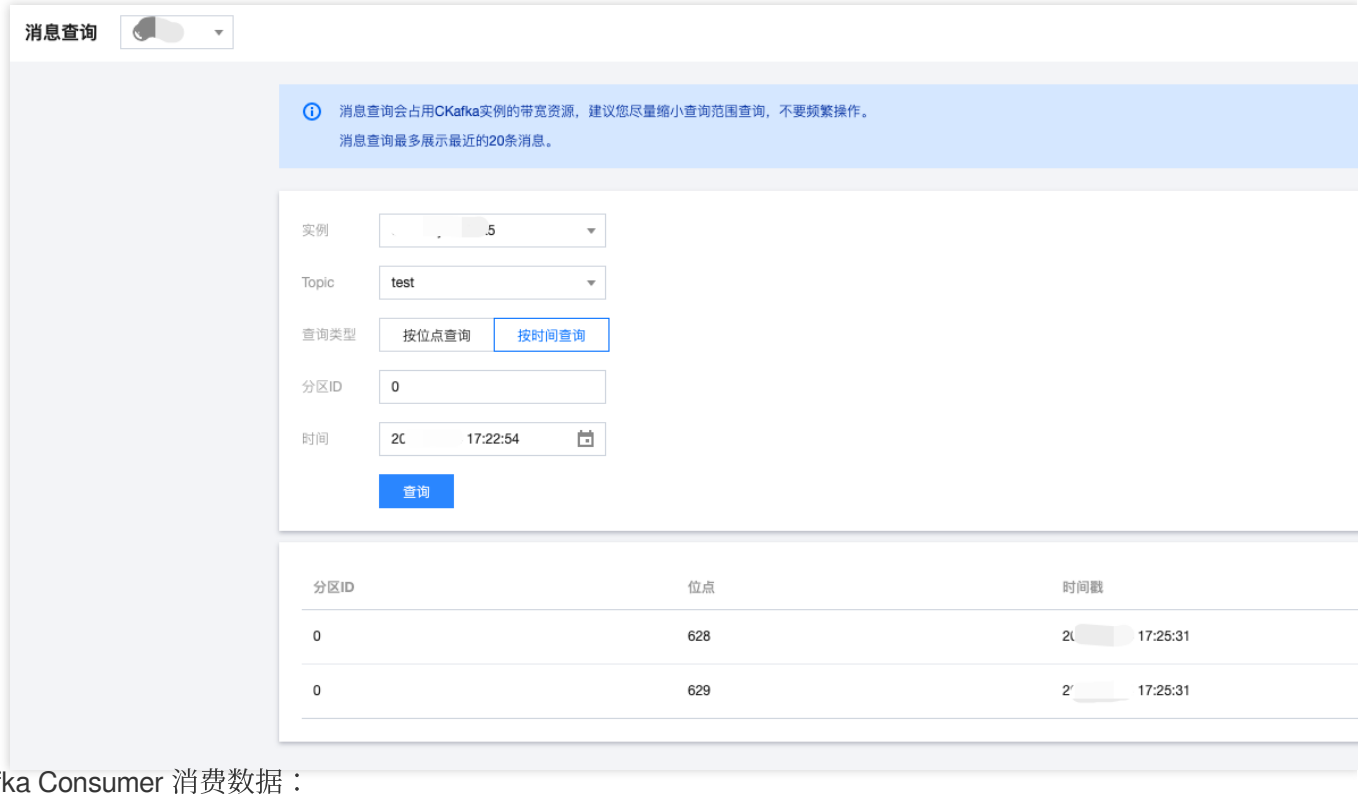
2. Kafka Producer 发送数据：



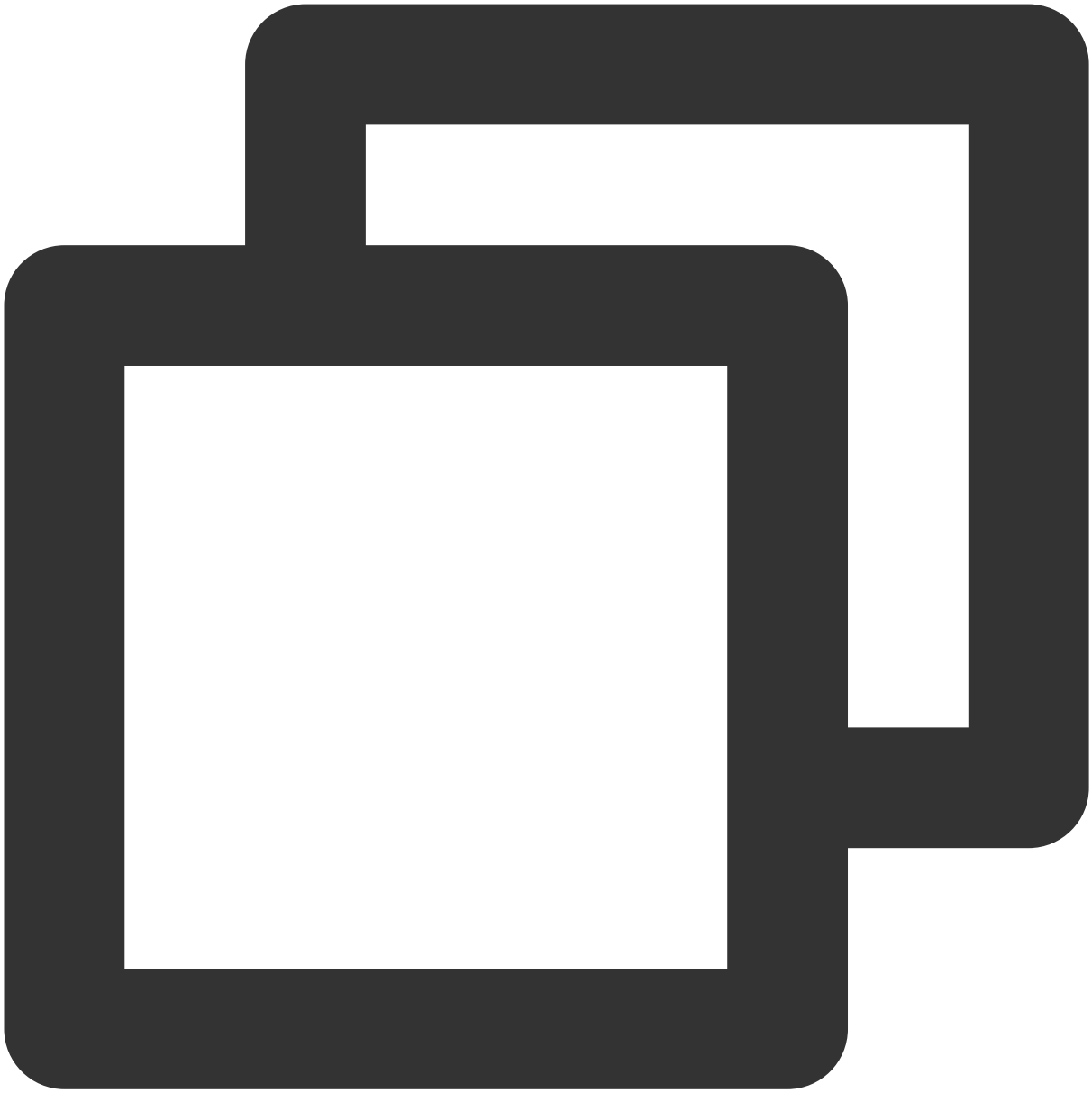
```
package schemaTest;
import java.util.Properties;
import java.util.Random;
import org.apache.avro.Schema;
import org.apache.avro.generic.GenericData;
import org.apache.avro.generic.GenericRecord;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerRecord;
public class SchemaProduce {
     public static final String USER_SCHEMA = "{\\"type\\": \\"record\\", \\"name\\
```

```
                "\\"fields\\": [{\\"name\\": \\"id\\", \\"type\\": \\"int\\"}, " +
                "{\\"name\\": \\"name\\",  \\"type\\": \\"string\\"}, {\\"name\\":
        public static void main(String[] args) throws Exception {
            Properties props = new Properties();
            // 添加CKafka实例的接入地址
          props.put("bootstrap.servers", "xx.xx.xx.xx:xxxx");
            props.put("key.serializer", "org.apache.kafka.common.serialization.Stri
            // 使用 Confluent 实现的 KafkaAvroSerializer
            props.put("value.serializer", "io.confluent.kafka.serializers.KafkaAvro
            // 添加 schema 服务的地址，用于获取 schema
            props.put("schema.registry.url", "http://127.0.0.1:8081");
            Producer<String, GenericRecord> producer = new KafkaProducer<>(props);
            Schema.Parser parser = new Schema.Parser();
            Schema schema = parser.parse(USER_SCHEMA);
            Random rand = new Random();
            int id = 0;
            while(id < 100) {
                id++;
                String name = "name" + id;
                int age = rand.nextInt(40) + 1;
                GenericRecord user = new GenericData.Record(schema);
                user.put("id", id);
                user.put("name", name);
                user.put("age", age);
                ProducerRecord<String, GenericRecord> record = new ProducerRecord<>
                producer.send(record);
                Thread.sleep(1000);
            }
            producer.close();
        }
}
```

运行一段时间后，在 CKafka 控制台 的**topic管理**页面，选择对应的 Topic ，单击**更多 > 消息查询**，查看刚刚发送的消息。



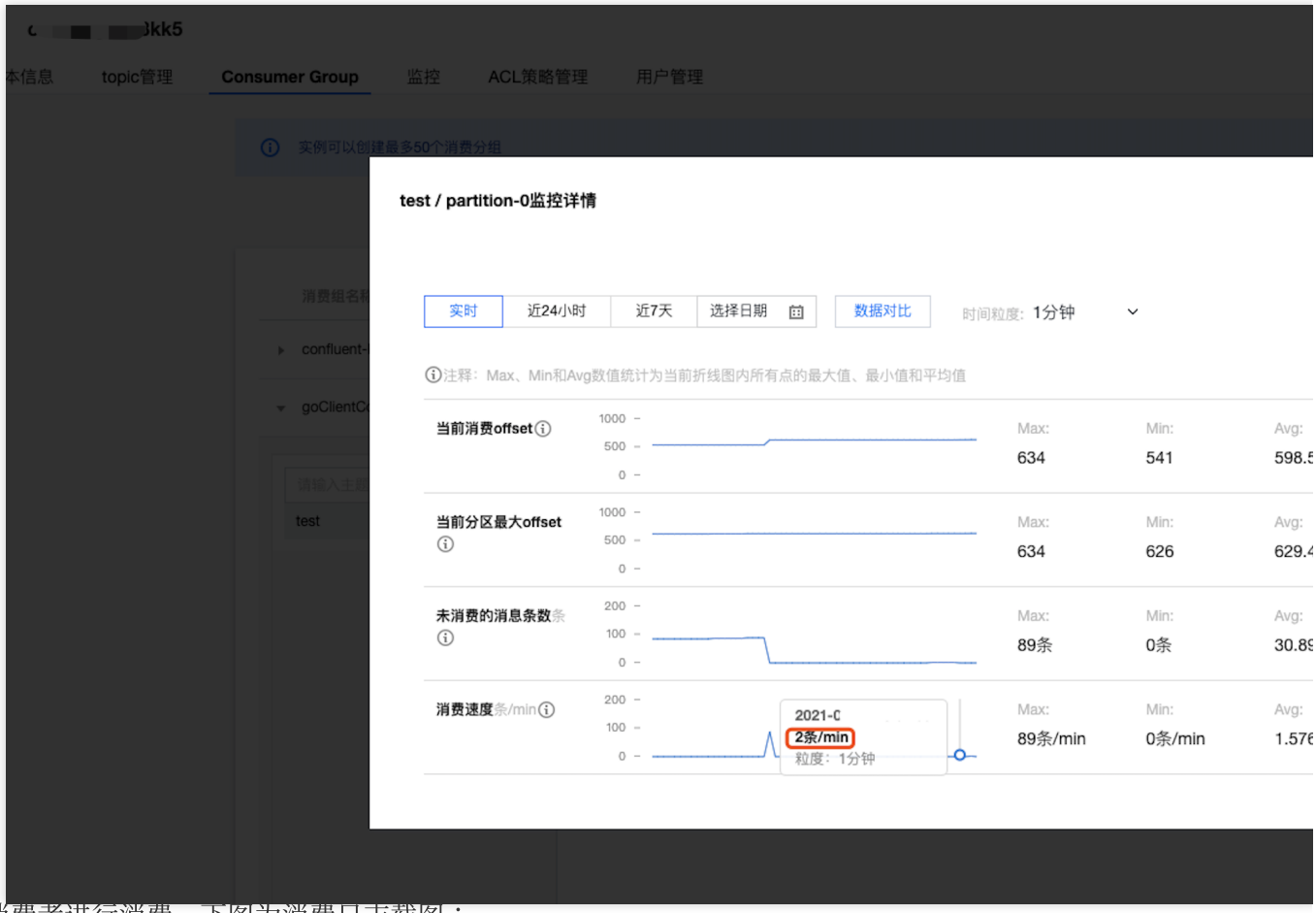
3. Kafka Consumer 消费数据：



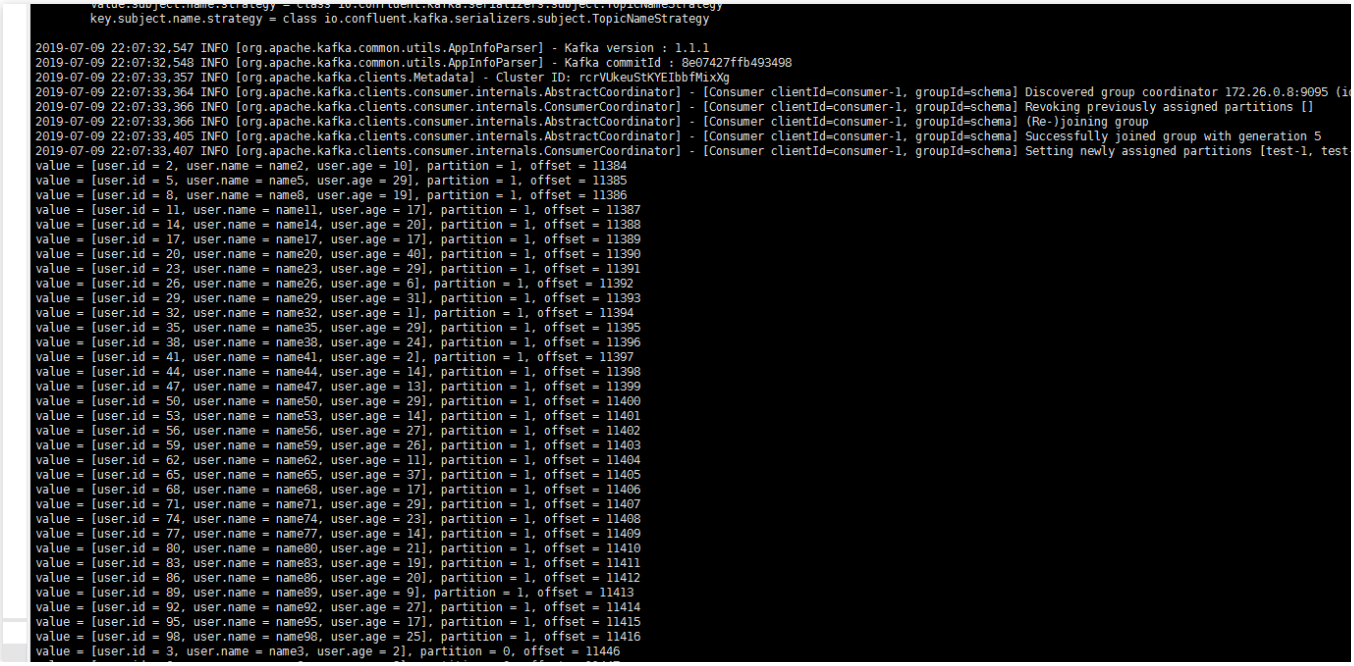
```
package schemaTest;
import java.util.Collections;
import java.util.Properties;
import org.apache.avro.generic.GenericRecord;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
public class SchemaProduce {
    public static void main(String[] args) throws Exception {
        Properties props = new Properties();
        props.put("bootstrap.servers", "xx.xx.xx.xx:xxxx"); //CKafka实例的接入地址
```

```
        props.put("group.id", "schema");
        props.put("key.deserializer", "org.apache.kafka.common.serialization.String
        // 使用Confluent实现的KafkaAvroDeserializer
        props.put("value.deserializer", "io.confluent.kafka.serializers.KafkaAvroDe
        // 添加schema服务的地址，用于获取schema
        props.put("schema.registry.url", "http://127.0.0.1:8081");
        KafkaConsumer<String, GenericRecord> consumer = new KafkaConsumer<>(props);
        consumer.subscribe(Collections.singletonList("test"));
        try {
            while (true) {
                ConsumerRecords<String, GenericRecord> records = consumer.poll(10);
                for (ConsumerRecord<String, GenericRecord> record : records) {
                    GenericRecord user = record.value();
                    System.out.println("value = [user.id = " + user.get("id") + ",
                            + user.get("name") + ", " + "user.age = " + user.get("a
                            + "partition = " + record.partition() + ", " + "offset
                }
            }
        } finally {
            consumer.close();
        }
    }
}
```

在 CKafka 控制台 的**Consumer Group**页面，选择 schema消费组名称，在主题名称输入 Topic 名称，单击**查看消费者详情**，查看消费详情。



启动消费者进行消费，下图为消费日志截图：

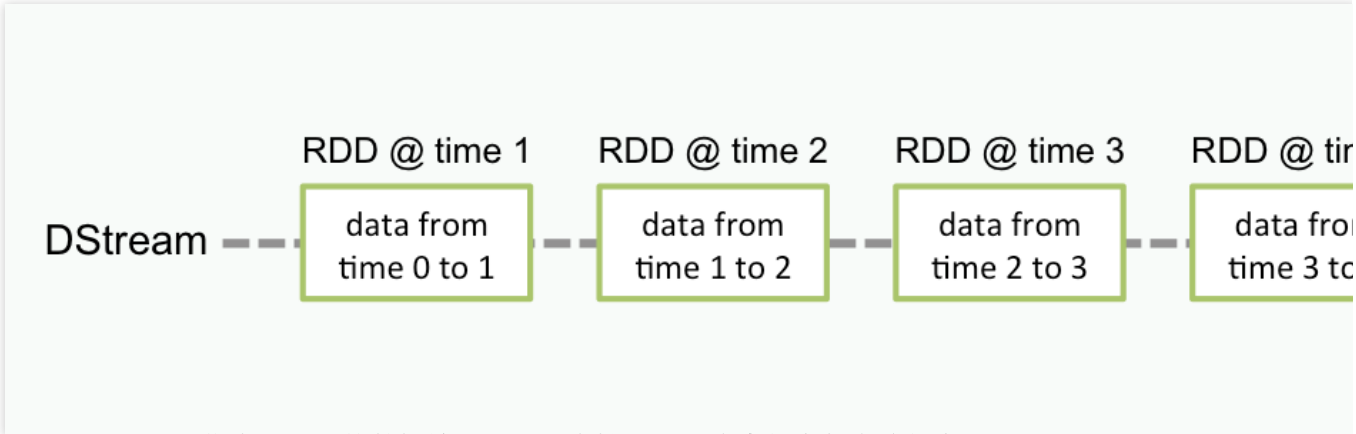


Spark Streaming 接入 CKafka

最近更新时间：2024-05-31 12:08:45

Spark Streaming 是 Spark Core 的一个扩展，用于高吞吐且容错地处理持续性的数据，目前支持的外部输入有 Kafka、Flume、HDFS/S3、Kinesis、Twitter 和 TCP socket。

Spark Streaming 将连续数据抽象成 DStream（Discretized Stream），而 DStream 由一系列连续的 RDD（弹性分布式数据集）组成，每个 RDD 是一定时间间隔内产生的数据。使用函数对 DStream 进行处理其实即为对这些 RDD 进行处理。



使用 Spark Streaming 作为 Kafka 的数据输入时，可支持 Kafka 稳定版本与实验版本：

Kafka Version	spark-streaming-kafka-0.8	spark-streaming-kafka-0.10
Broker Version	0.8.2.1 or higher	0.10.0 or higher
Api Maturity	Deprecated	Stable
Language Support	Scala、Java、Python	Scala、Java
Receiver DStream	Yes	No
Direct DStream	Yes	Yes

SSL / TLS Support	No	Yes
Offset Commit Api	No	Yes
Dynamic Topic Subscription	No	Yes

目前 CKafka 兼容 0.9及以上的版本，本次实践使用 0.10.2.1 版本的 Kafka 依赖。
此外，EMR 中的 Spark Streaming 也支持直接对接 CKafka，详见 SparkStreaming 对接 CKafka 服务。

操作步骤

步骤1：获取 CKafka 实例接入地址

1. 登录 CKafka 控制台。
2. 在左侧导航栏选择**实例列表**，单击实例的“ID”，进入实例基本信息页面。
3. 在实例的基本信息页面的**接入方式**模块，可获取实例的接入地址，接入地址是生产消费需要用到的 bootstrap-server。

步骤2：创建 Topic

1. 在实例基本信息页面，选择顶部**Topic管理**页签。
2. 在 Topic 管理页面，单击**新建**，创建一个名为 test 的 Topic，接下来将以该 Topic 为例介绍如何生产消费。

步骤3：准备云服务器环境

Centos6.8 系统

package	version
sbt	0.13.16

hadoop	2.7.3
spark	2.1.0
protobuf	2.5.0
ssh	CentOS 默认安装
Java	1.8

具体安装步骤参见 配置环境。

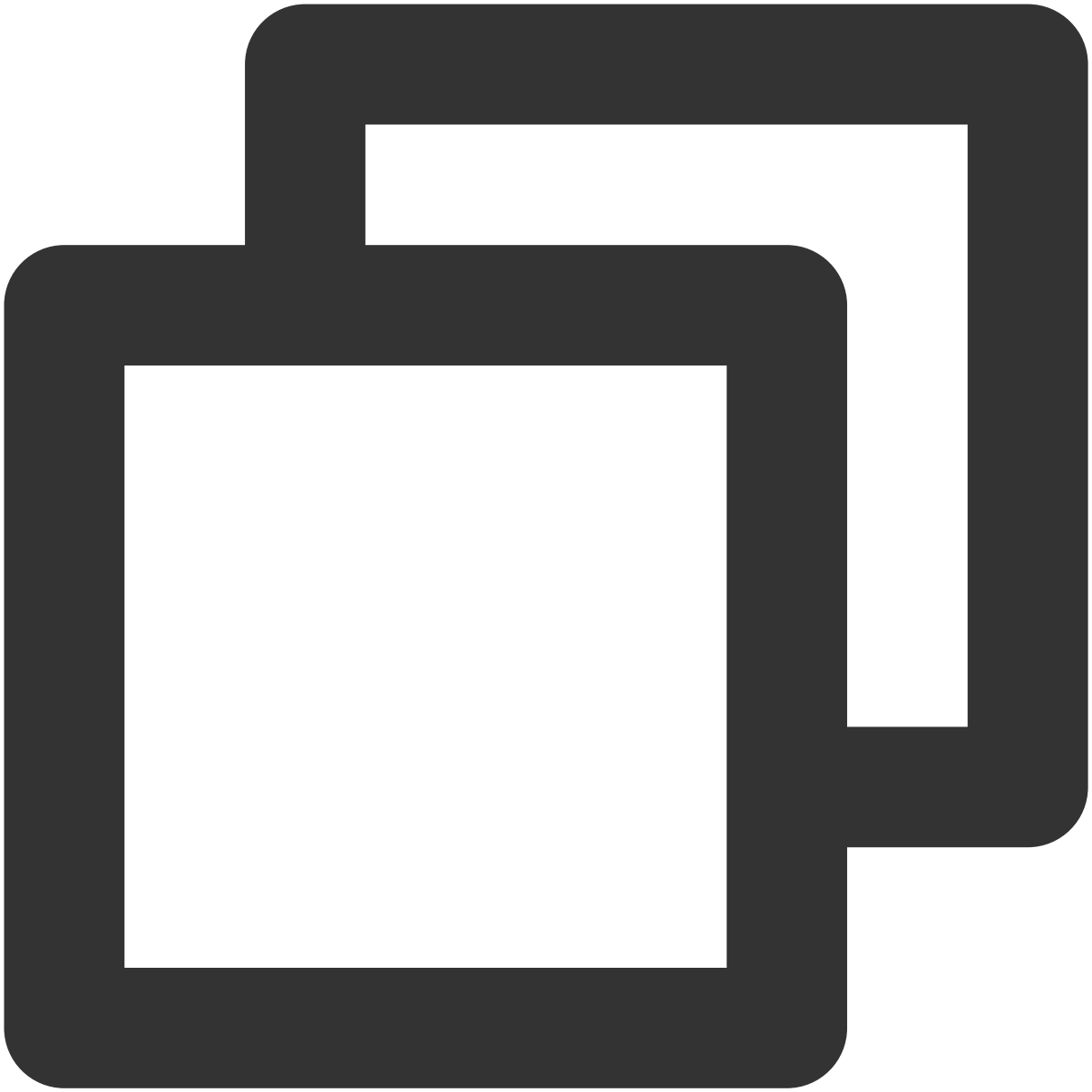
步骤4：对接 CKafka

向 CKafka 中生产消息

从 CKafka 消费消息

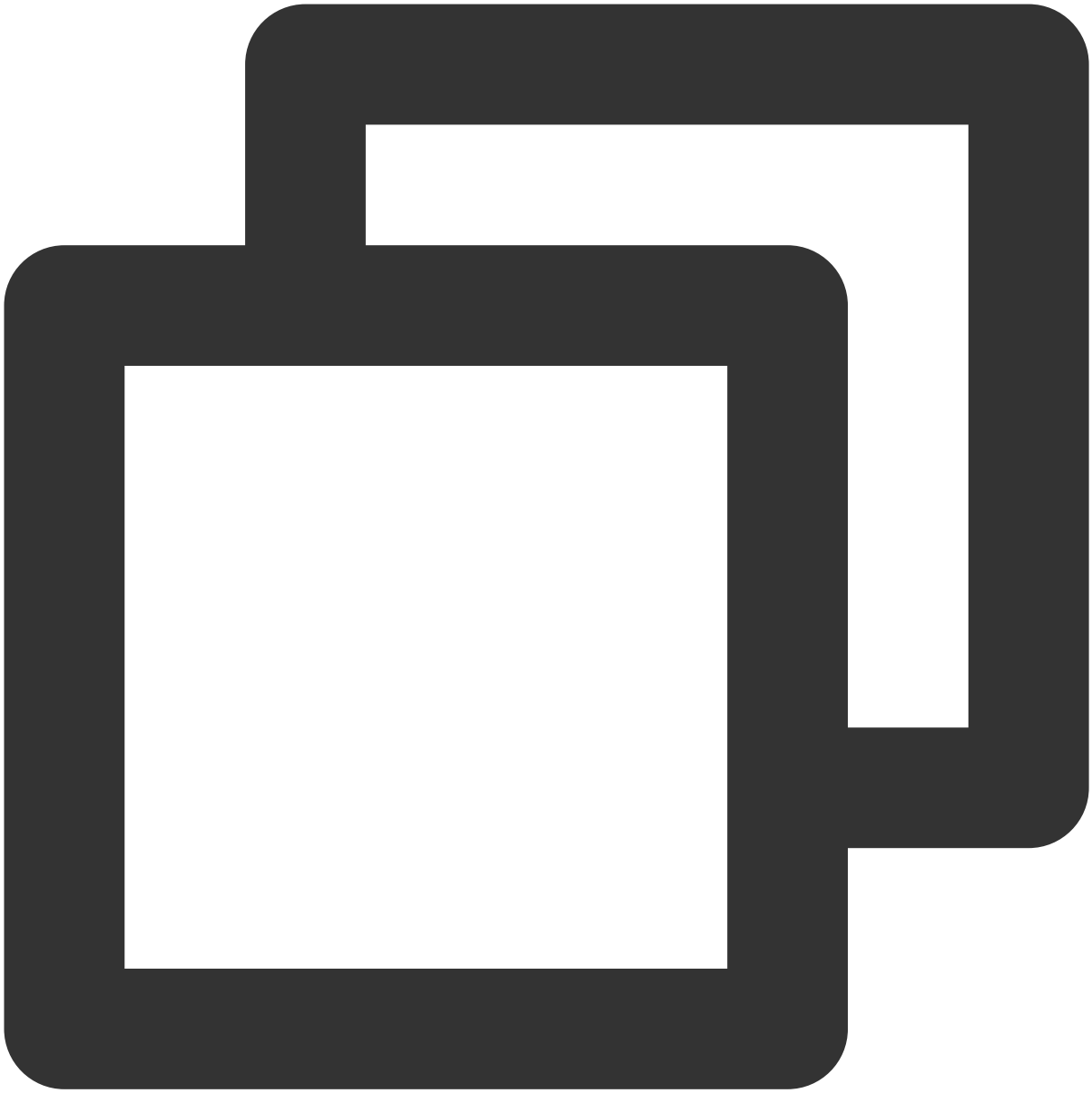
这里使用 0.10.2.1 版本的 Kafka 依赖。

1. 在 `build.sbt` 添加依赖：



```
name := "Producer Example"
version := "1.0"
scalaVersion := "2.11.8"
libraryDependencies += "org.apache.kafka" % "kafka-clients" % "0.10.2.1"
```

2. 配置 `producer_example.scala` ：



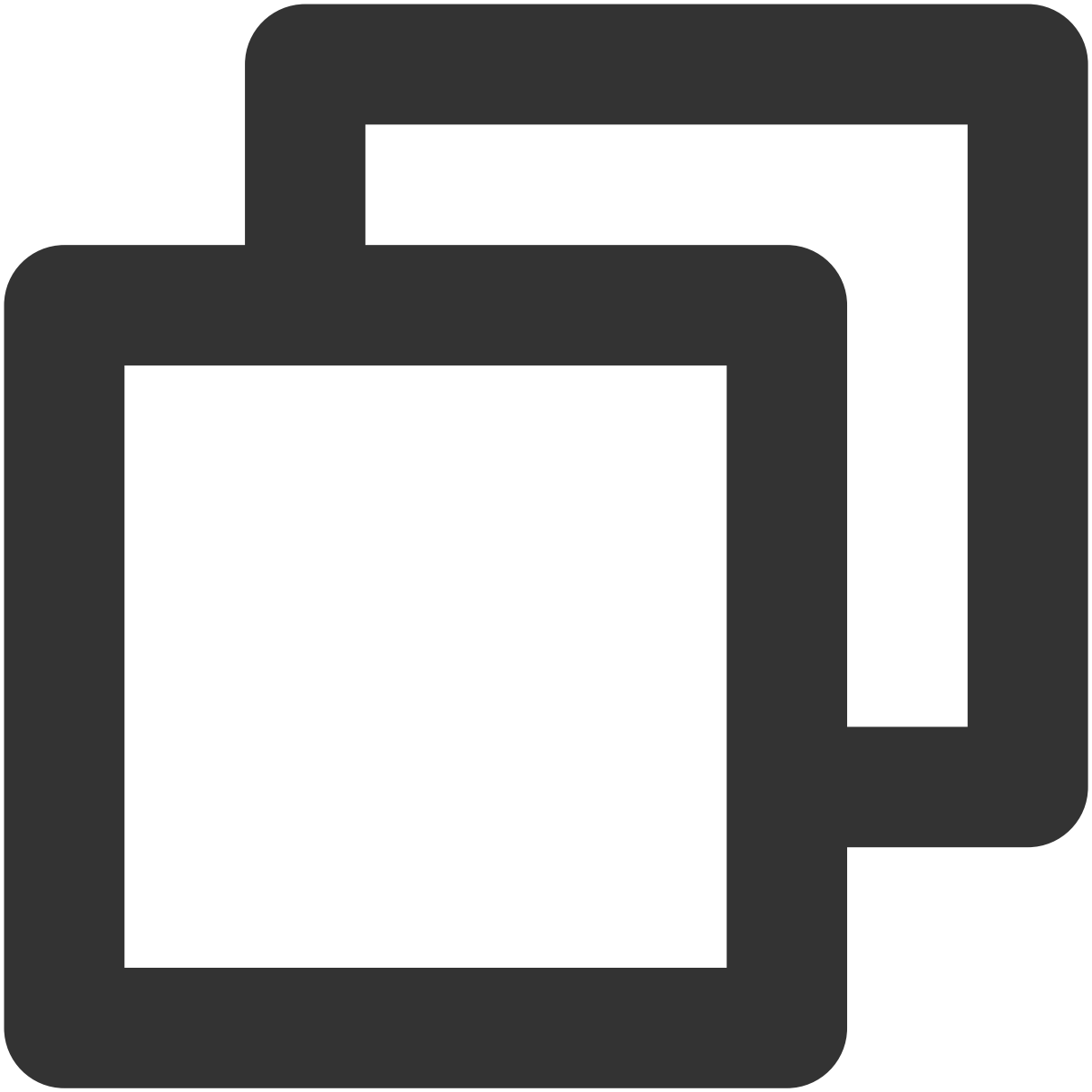
```
import java.util.Properties
import org.apache.kafka.clients.producer._
object ProducerExample extends App {
 val  props = new Properties()
 props.put("bootstrap.servers", "172.16.16.12:9092") //实例信息中的内网 IP 与端口

 props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerial
 props.put("value.serializer", "org.apache.kafka.common.serialization.StringSeri

 val producer = new KafkaProducer[String, String](props)
 val TOPIC="test"  //指定要生产的 Topic
```

```
    for(i<- 1 to 50){
           val record = new ProducerRecord(TOPIC, "key", s"hello $i") //生产 key 是"
           producer.send(record)
    }
    val record = new ProducerRecord(TOPIC, "key", "the end "+new java.util.Date)
    producer.send(record)
    producer.close() //最后要断开
   }
```

更多有关 ProducerRecord 的用法请参见 ProducerRecord 文档。

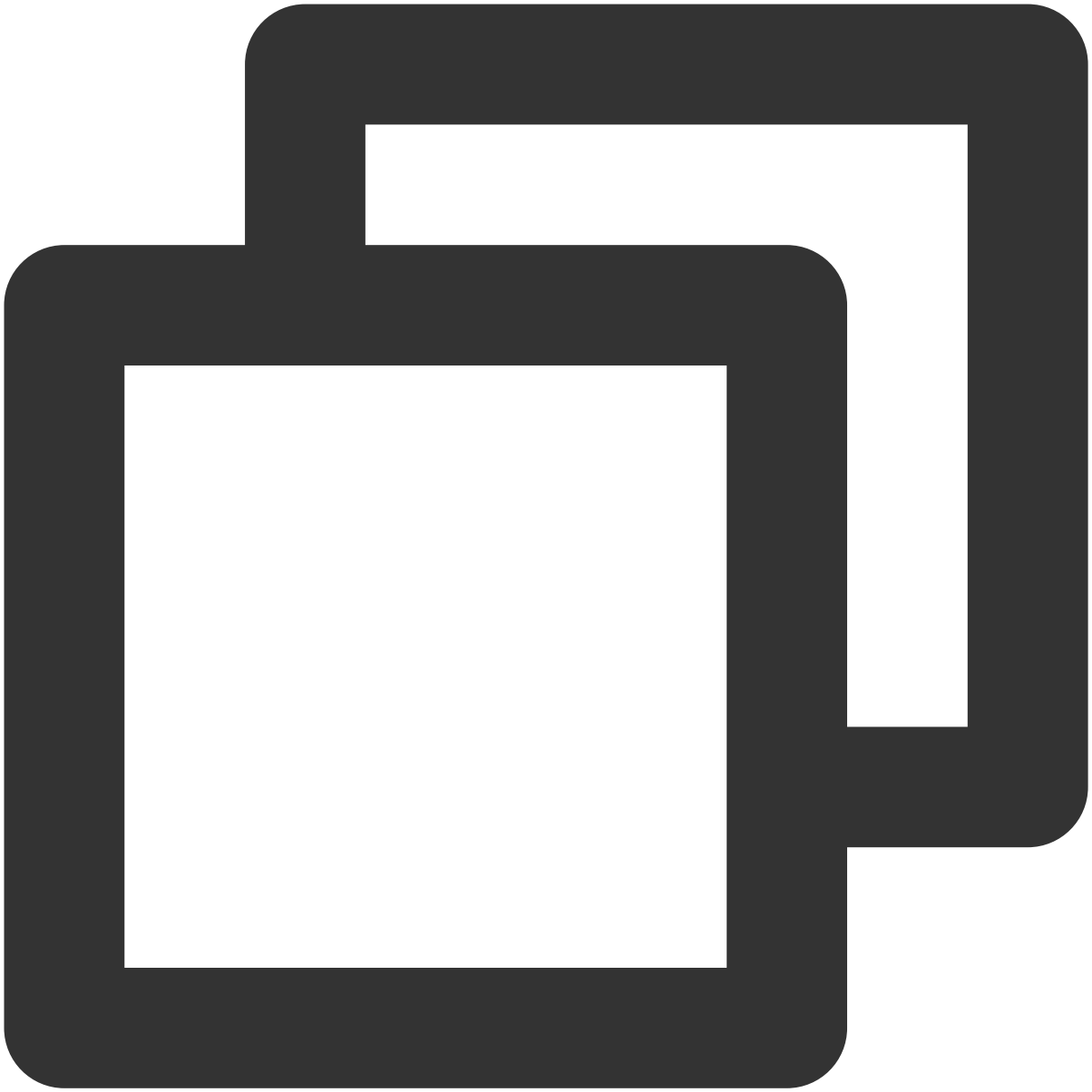
DirectStream

1. 在 `build.sbt` 添加依赖：



```
name := "Consumer Example"
version := "1.0"
scalaVersion := "2.11.8"
libraryDependencies += "org.apache.spark" %% "spark-core" % "2.1.0"
libraryDependencies += "org.apache.spark" %% "spark-streaming" % "2.1.0"
libraryDependencies += "org.apache.spark" %% "spark-streaming-kafka-0-10" % "2.1.0"
```

2. 配置 `DirectStream_example.scala` ：



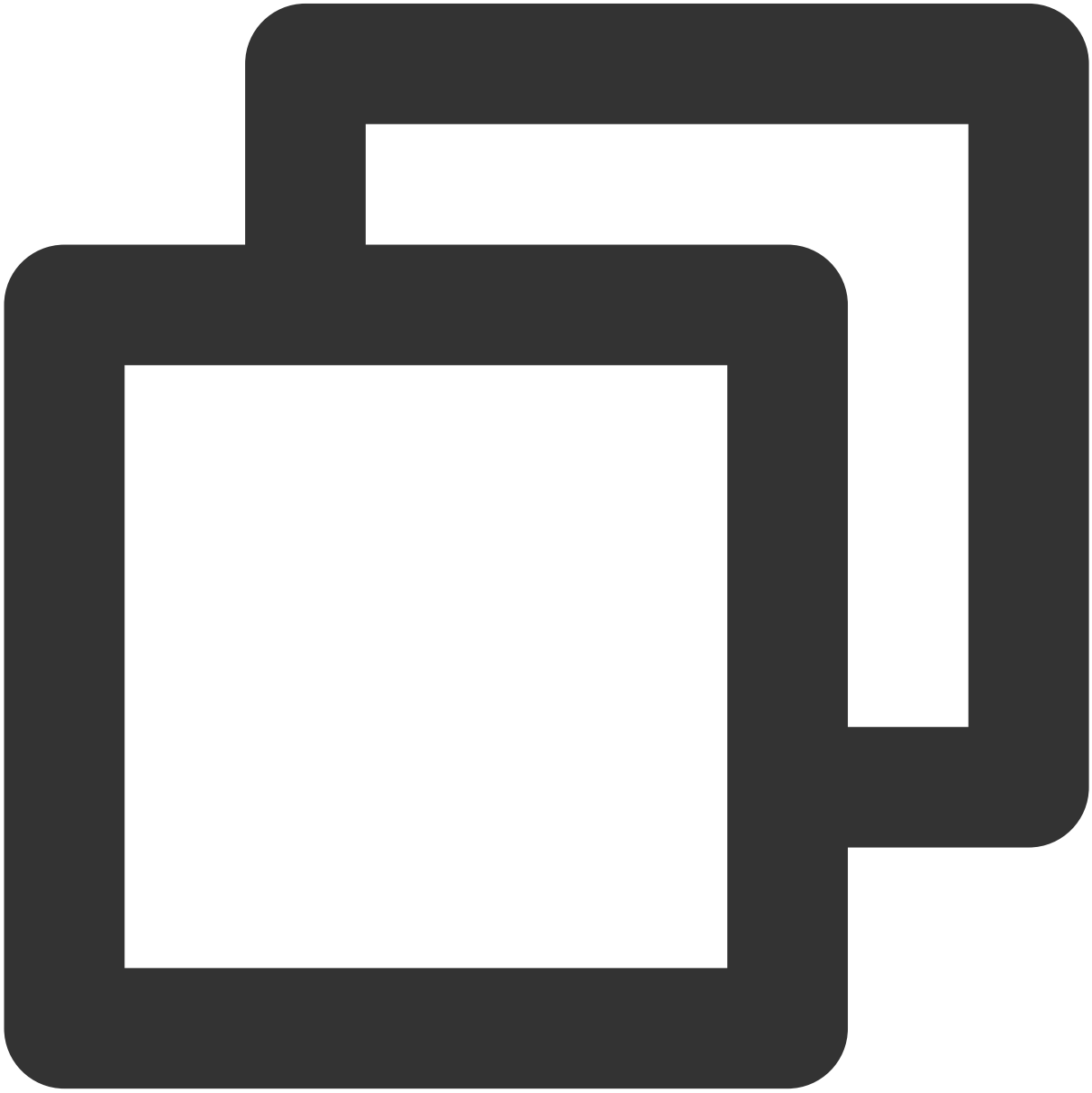
```
import org.apache.kafka.clients.consumer.ConsumerRecord
import org.apache.kafka.common.serialization.StringDeserializer
import org.apache.kafka.common.TopicPartition
import org.apache.spark.streaming.kafka010._
import org.apache.spark.streaming.kafka010.LocationStrategies.PreferConsistent
import org.apache.spark.streaming.kafka010.ConsumerStrategies.Subscribe
import org.apache.spark.streaming.kafka010.KafkaUtils
import org.apache.spark.streaming.kafka010.OffsetRange
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
```

```
import collection.JavaConversions._
import Array._
object Kafka {
    def main(args: Array[String]) {
        val kafkaParams = Map[String, Object](
            "bootstrap.servers" -> "172.16.16.12:9092",
            "key.deserializer" -> classOf[StringDeserializer],
            "value.deserializer" -> classOf[StringDeserializer],
            "group.id" -> "spark_stream_test1",
            "auto.offset.reset" -> "earliest",
            "enable.auto.commit" -> "false"
        )

        val sparkConf = new SparkConf()
        sparkConf.setMaster("local")
        sparkConf.setAppName("Kafka")
        val ssc = new StreamingContext(sparkConf, Seconds(5))
        val topics = Array("spark_test")

        val offsets : Map[TopicPartition, Long] = Map()

        for (i <- 0 until 3){
            val tp = new TopicPartition("spark_test", i)
            offsets.updated(tp , 0L)
        }
        val stream = KafkaUtils.createDirectStream[String, String](
            ssc,
            PreferConsistent,
            Subscribe[String, String](topics, kafkaParams)
        )
        println("directStream")
        stream.foreachRDD{ rdd=>
            //输出获得的消息
            rdd.foreach{iter =>
                val i = iter.value
                println(s"${i}")
            }
            //获得offset
            val offsetRanges = rdd.asInstanceOf[HasOffsetRanges].offsetRanges
            rdd.foreachPartition { iter =>
                val o: OffsetRange = offsetRanges(TaskContext.get.partitionId)
                println(s"${o.topic} ${o.partition} ${o.fromOffset} ${o.untilOffset
            }
        }

        // Start the computation
        ssc.start()
```

```
        ssc.awaitTermination()
    }
}
```

RDD

1. 配置 `build.sbt` （配置同上，单击查看）。
2. 配置 `RDD_example` ：



```
import org.apache.kafka.clients.consumer.ConsumerRecord
```

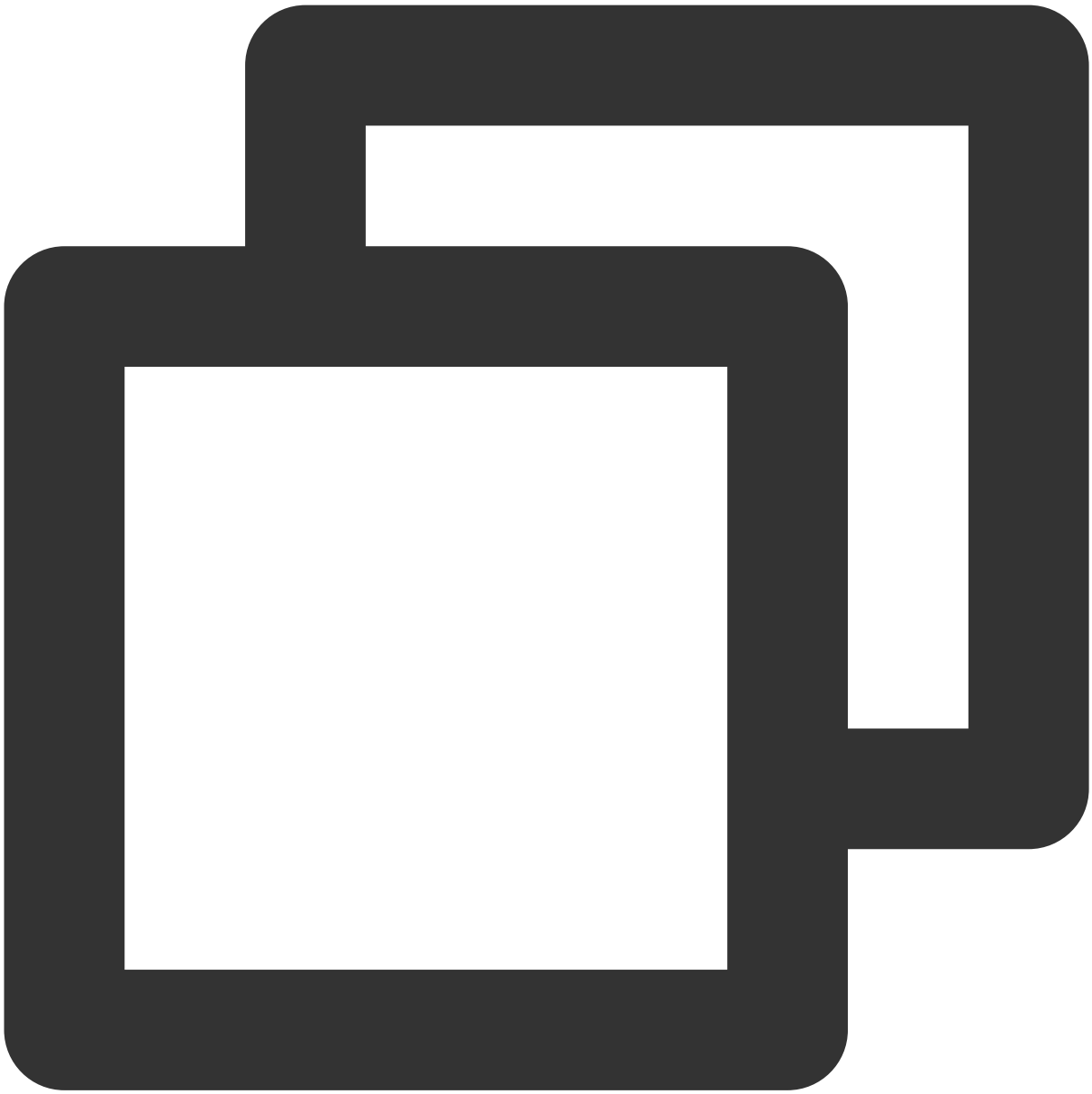
```
import org.apache.kafka.common.serialization.StringDeserializer
import org.apache.spark.streaming.kafka010._
import org.apache.spark.streaming.kafka010.LocationStrategies.PreferConsistent
import org.apache.spark.streaming.kafka010.ConsumerStrategies.Subscribe
import org.apache.spark.streaming.kafka010.KafkaUtils
import org.apache.spark.streaming.kafka010.OffsetRange
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import collection.JavaConversions._
import Array._
object Kafka {
    def main(args: Array[String]) {
        val kafkaParams = Map[String, Object](
            "bootstrap.servers" -> "172.16.16.12:9092",
            "key.deserializer" -> classOf[StringDeserializer],
            "value.deserializer" -> classOf[StringDeserializer],
            "group.id" -> "spark_stream",
            "auto.offset.reset" -> "earliest",
            "enable.auto.commit" -> (false: java.lang.Boolean)
        )
        val sc = new SparkContext("local", "Kafka", new SparkConf())
        val java_kafkaParams : java.util.Map[String, Object] = kafkaParams
        //按顺序向 parition 拉取相应 offset 范围的消息，如果拉取不到则阻塞直到超过等待时间或者
        val offsetRanges = Array[OffsetRange](
            OffsetRange("spark_test", 0, 0, 5),
            OffsetRange("spark_test", 1, 0, 5),
            OffsetRange("spark_test", 2, 0, 5)
        )
        val range = KafkaUtils.createRDD[String, String](
            sc,
            java_kafkaParams,
            offsetRanges,
            PreferConsistent
        )
        range.foreach(rdd=>println(rdd.value))
        sc.stop()
    }
}
```

更多 `kafkaParams` 用法参见 kafkaParams 文档。

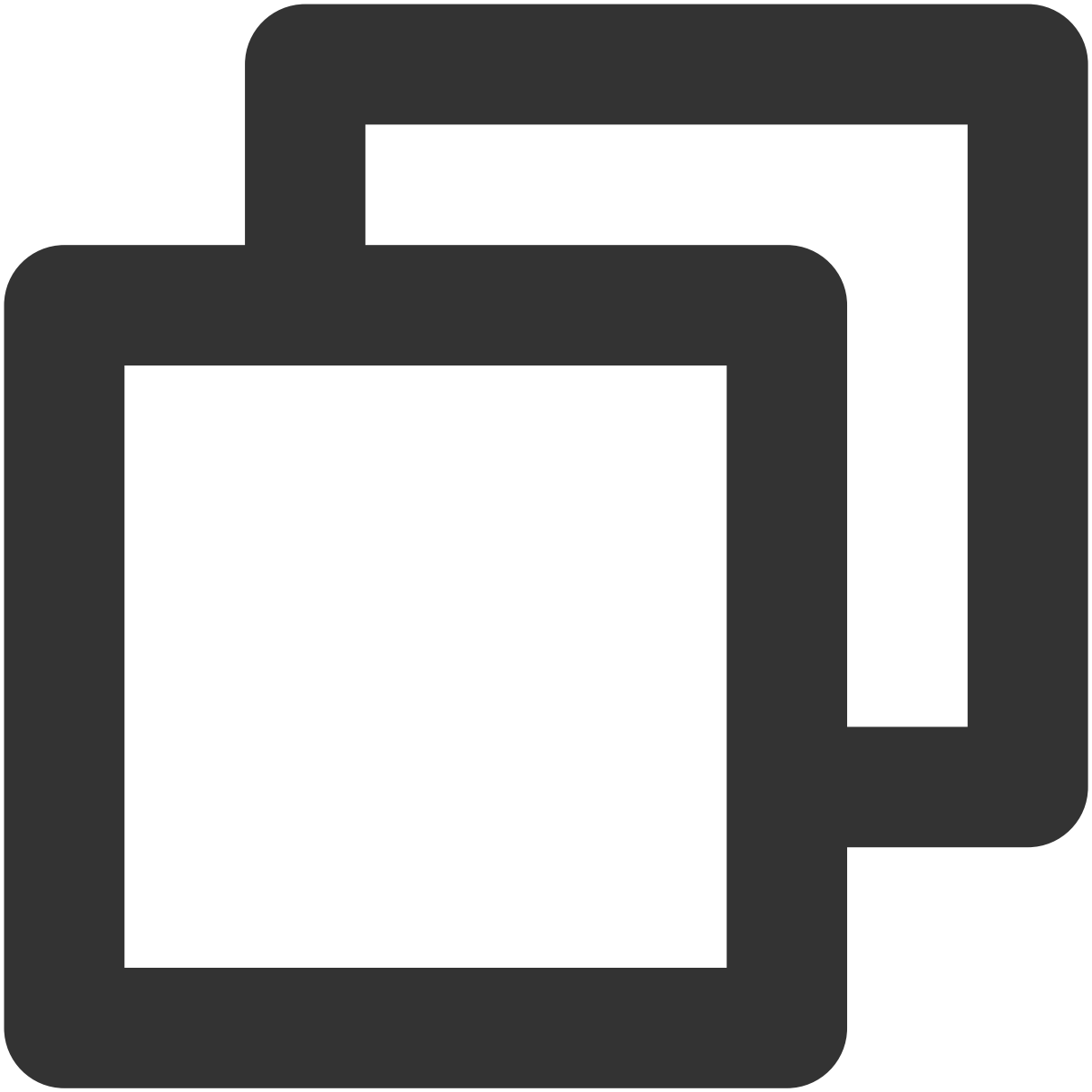
配置环境

安装 sbt

1. 在 sbt 官网 上下载 sbt 包。
2. 解压后在 sbt 的目录下创建一个 sbt_run.sh 脚本并增加可执行权限，脚本内容如下：

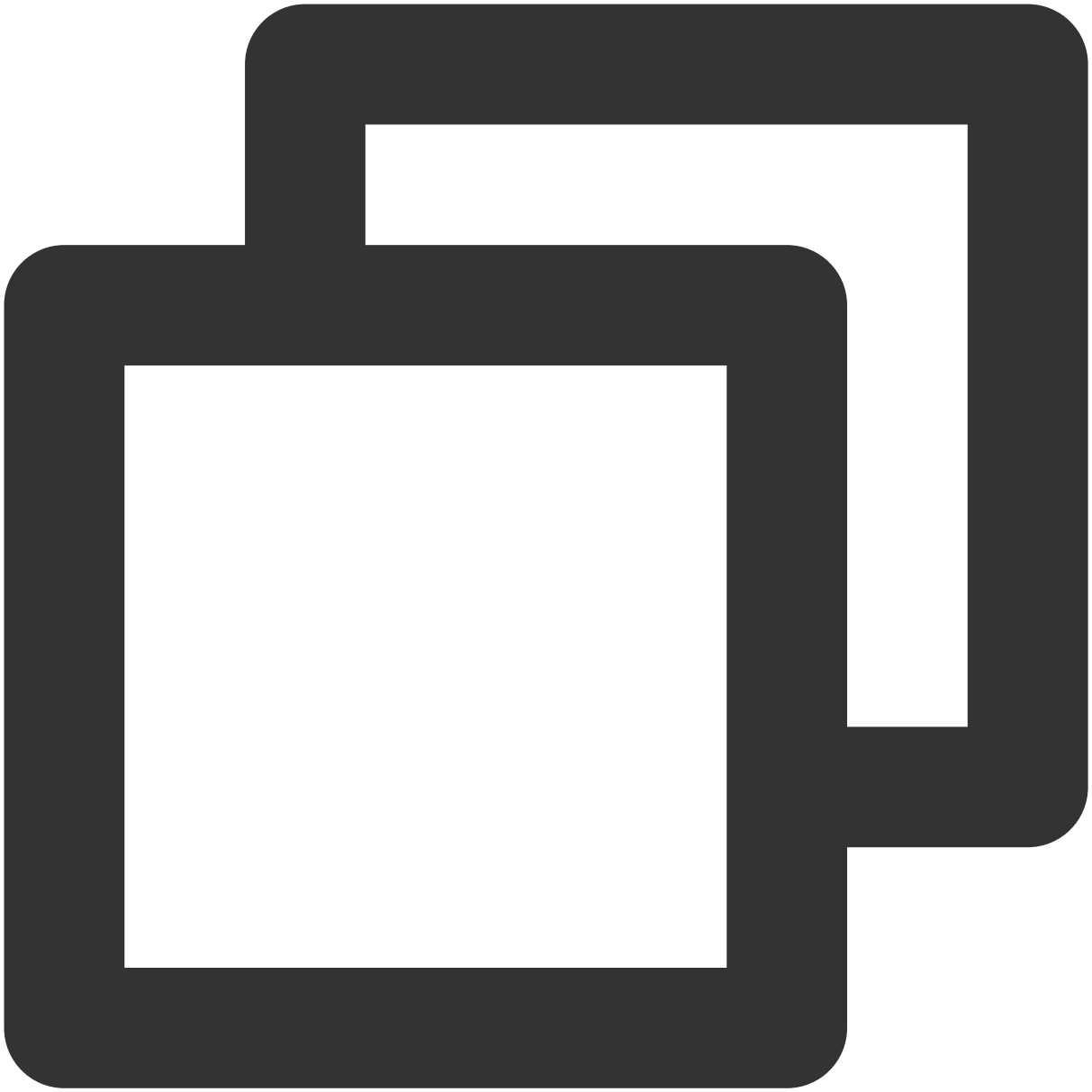


```
#!/bin/bash
SBT_OPTS="-Xms512M -Xmx1536M -Xss1M -XX:+CMSClassUnloadingEnabled -XX:MaxPermSize=2
java $SBT_OPTS -jar `dirname $0`/bin/sbt-launch.jar "$@"
```



```
chmod u+x ./sbt_run.sh
```

3. 执行以下命令。

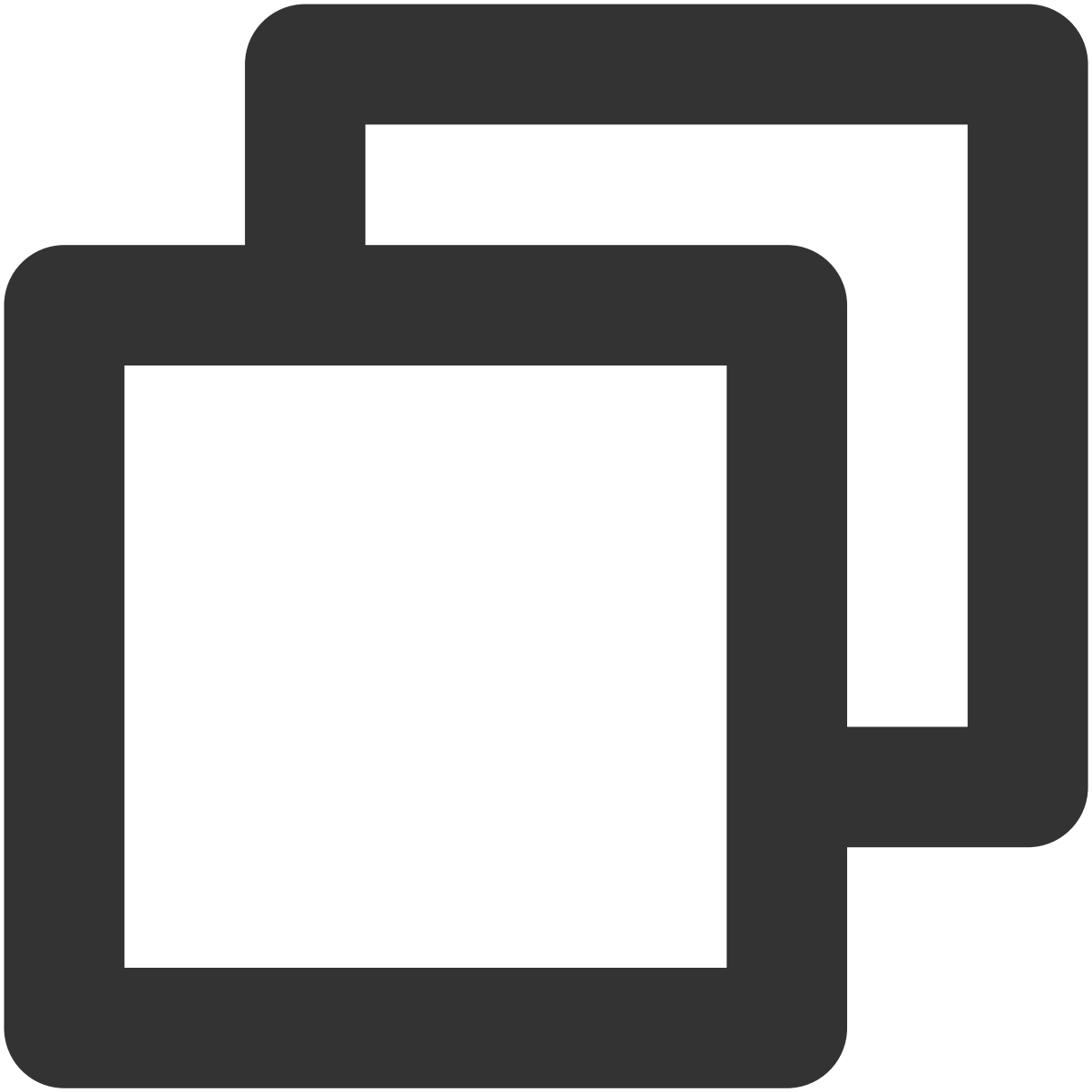


```
./sbt-run.sh sbt-version
```

若能看到 sbt 版本说明可以正常运行。

安装 protobuf

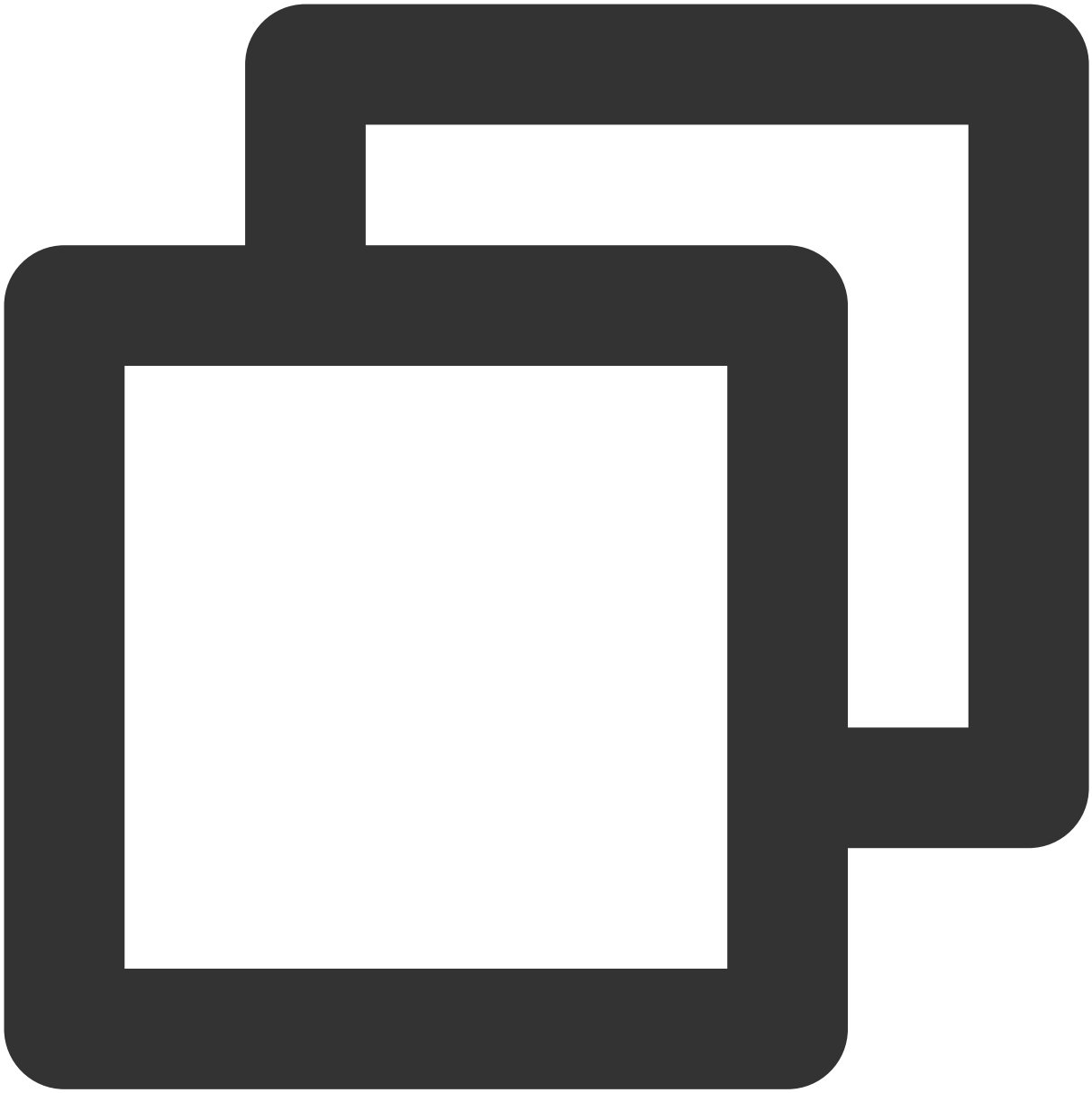
1. 下载 protobuf 相应版本。
2. 解压后进入目录。



```
./configure
make && make install
```

需要预先安装 gcc-g++，执行中可能需要 root 权限。

3. 重新登录，在命令行中输入下述内容。

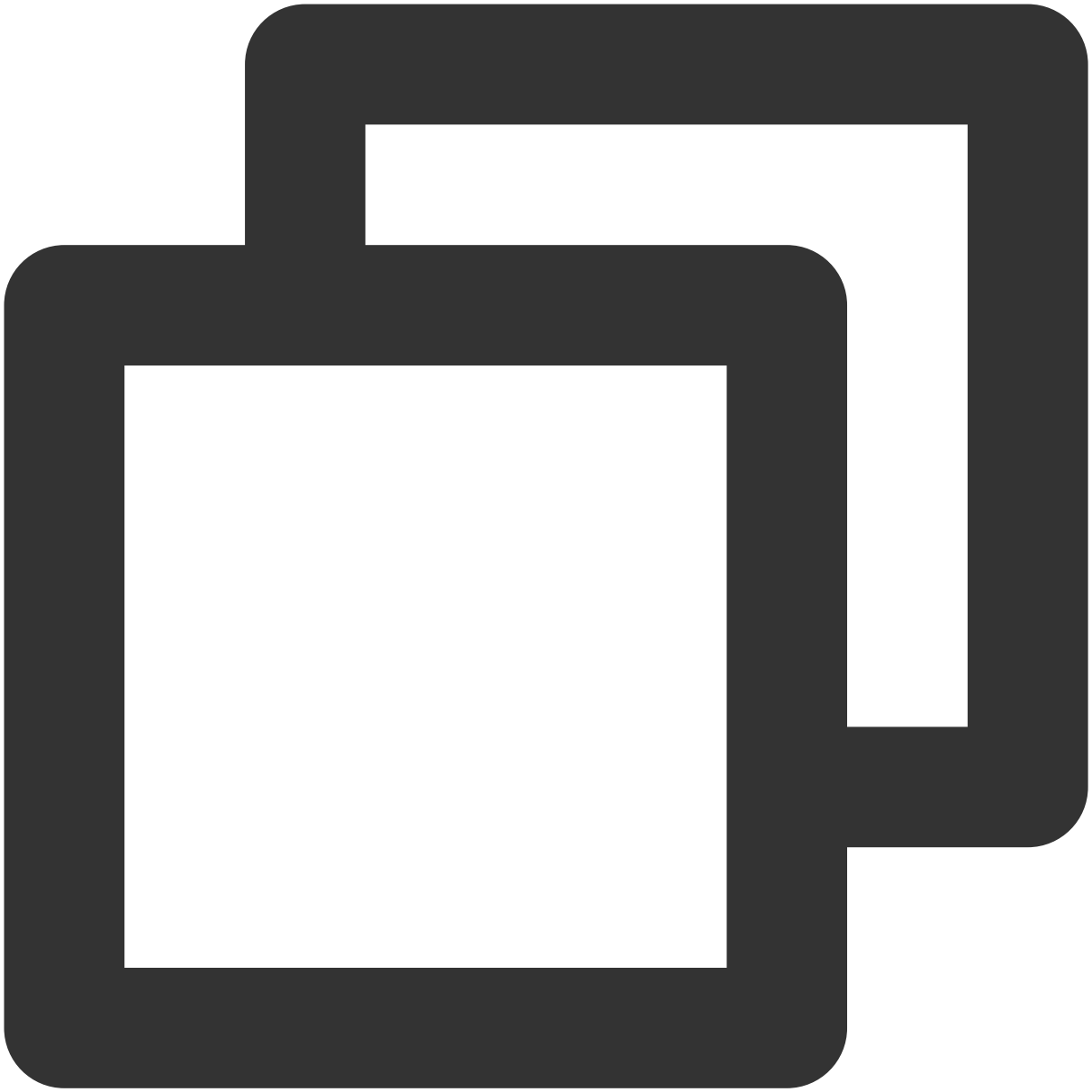


```
protoc --version
```

4. 若能看到 protobuf 版本说明可以正常运行。

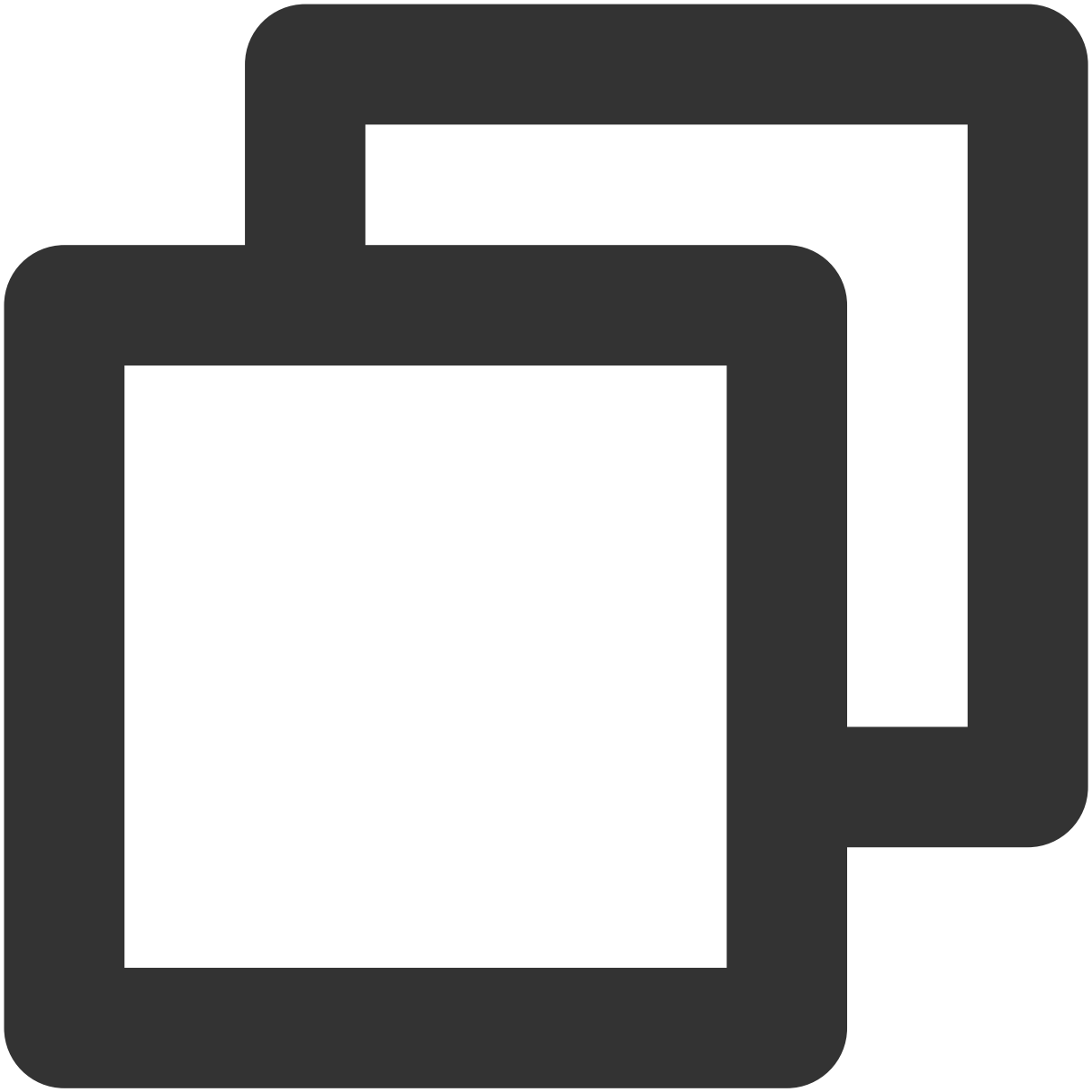
安装 Hadoop

1. 访问 Hadoop 官网 下载所需要的版本。
2. 增加 Hadoop 用户。



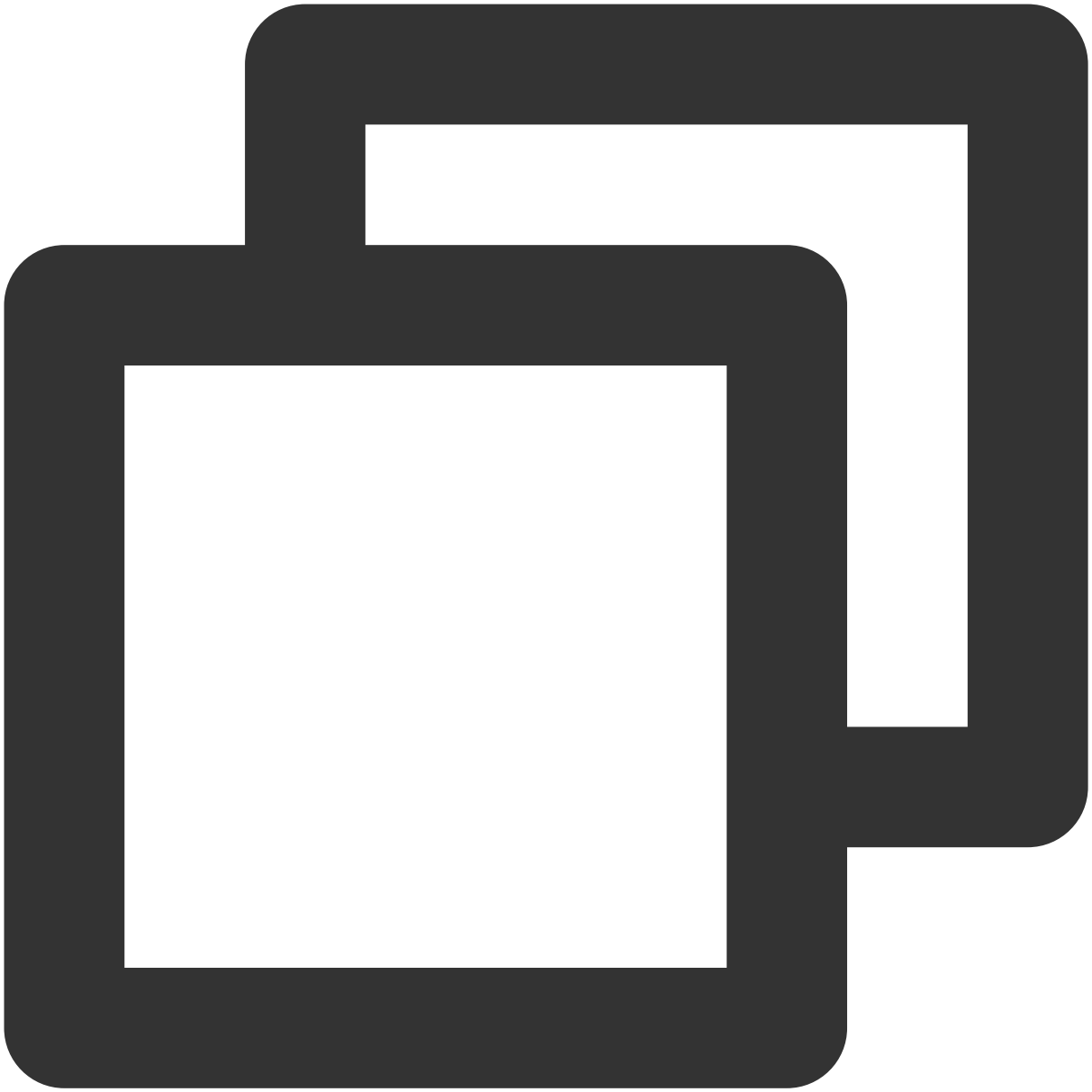
```
useradd -m hadoop -s /bin/bash
```

3. 增加管理员权限。



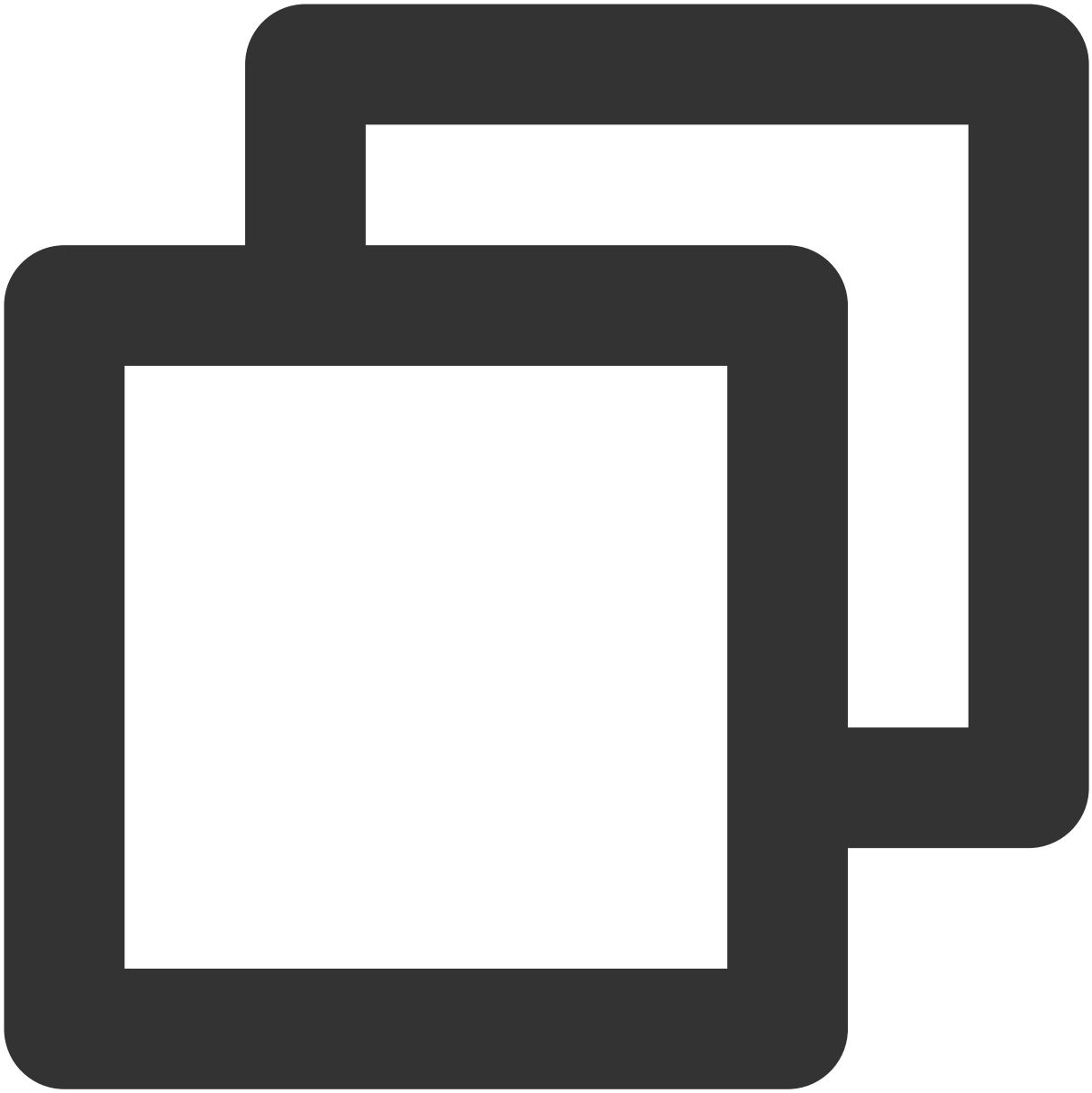
```
visudo
```

4. 在 `root ALL=(ALL) ALL` 下增加一行。 `hadoop ALL=(ALL) ALL` 保存退出。
5. 使用 Hadoop 进行操作。



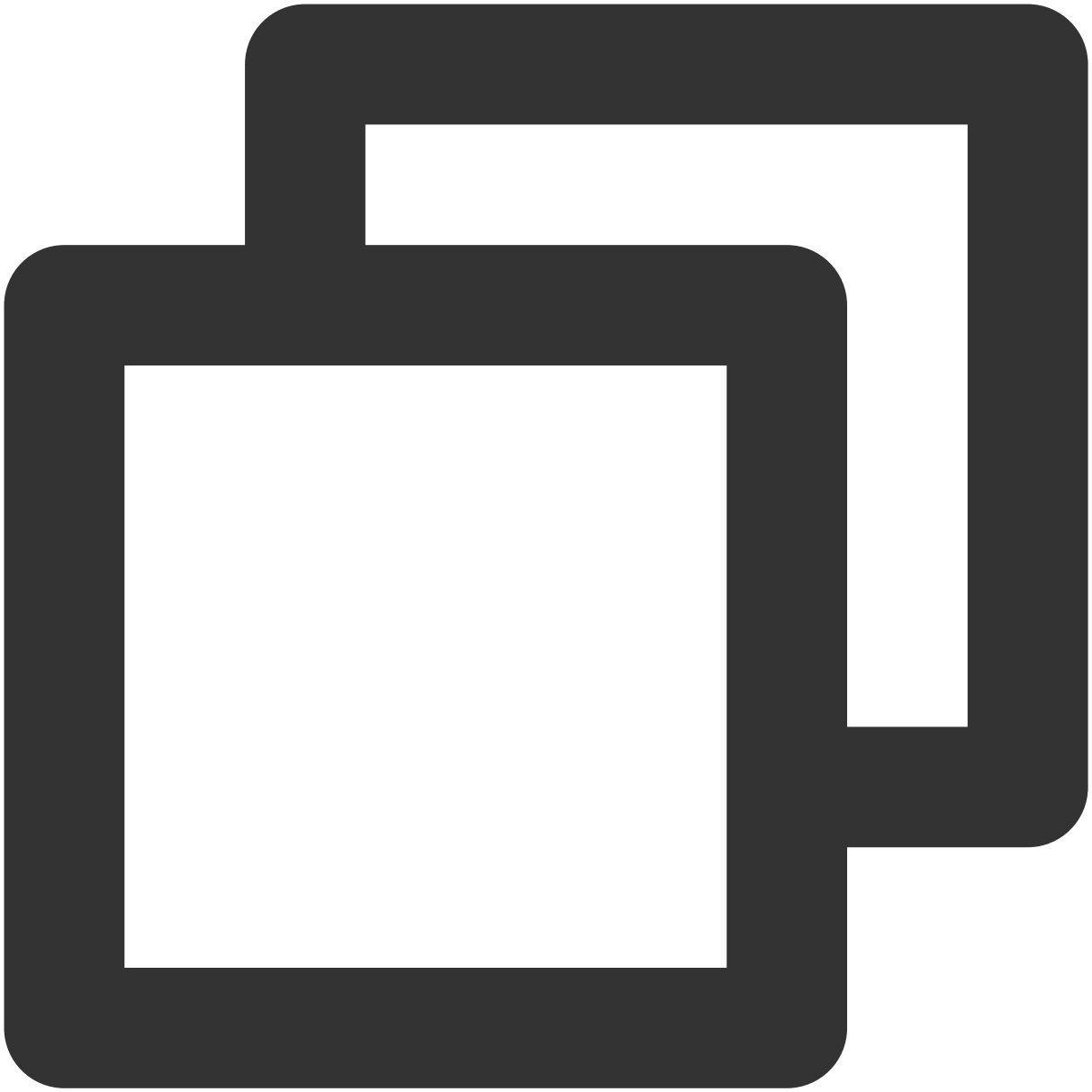
```
su hadoop
```

6. SSH 无密码登录。



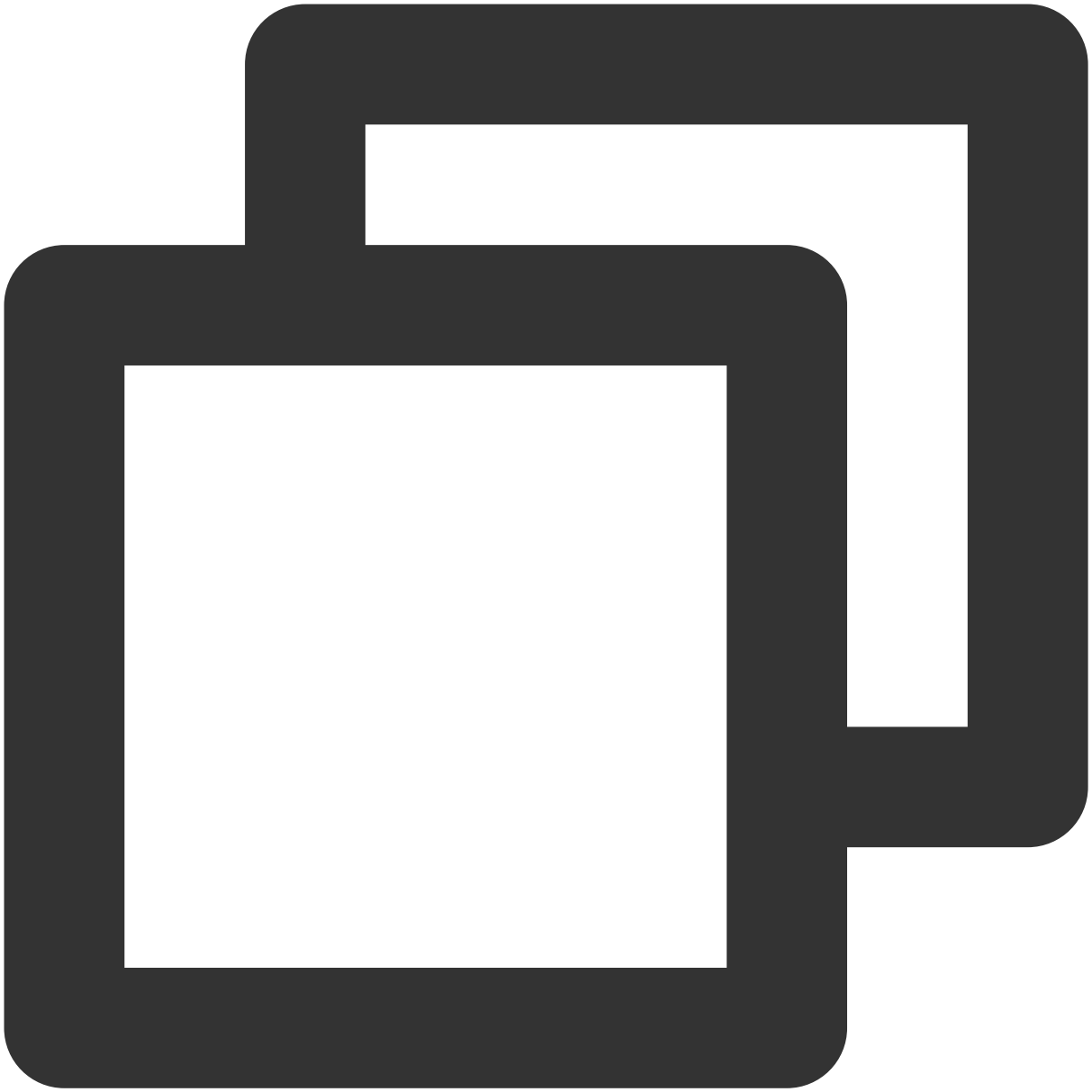
```
cd ~/.ssh/                          # 若没有该目录，请先执行一次ssh localhost
ssh-keygen -t rsa                   # 会有提示，都按回车就可以
cat id_rsa.pub >> authorized_keys   # 加入授权
chmod 600 ./authorized_keys         # 修改文件权限
```

7. 安装 Java。



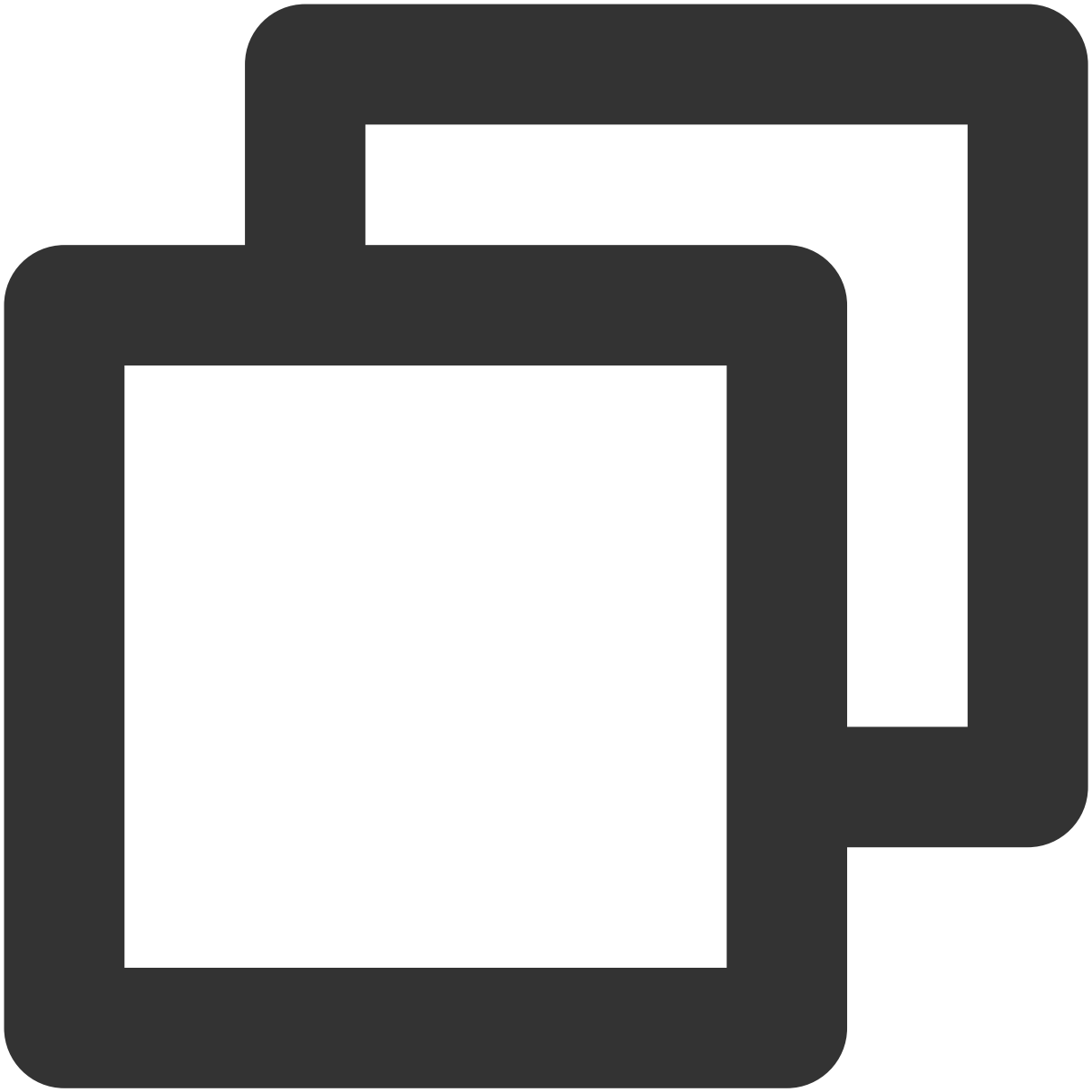
```
sudo yum install java-1.8.0-openjdk java-1.8.0-openjdk-devel
```

8. 配置 ${JAVA_HOME}。



```
vim /etc/profile
```

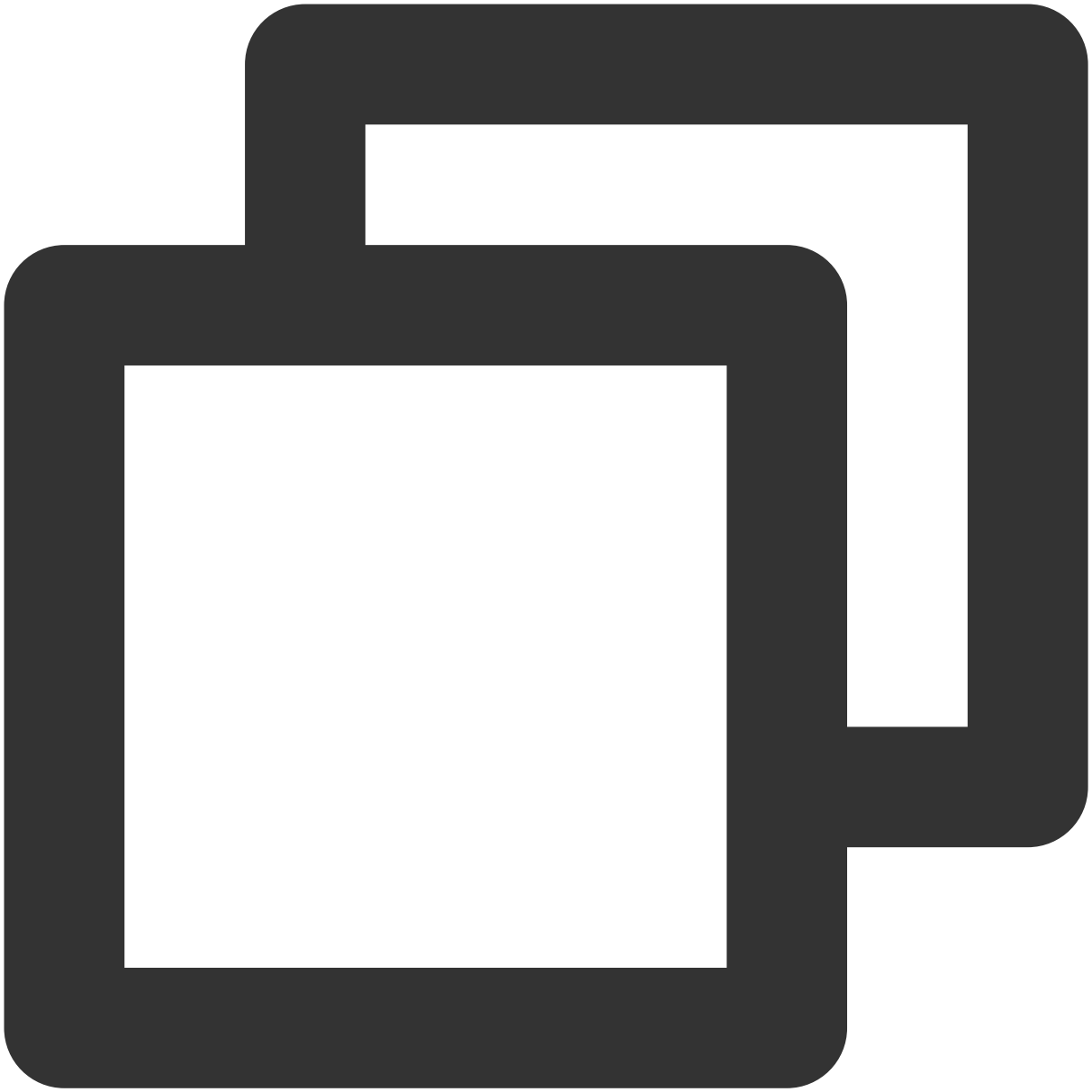
在文末加上下述内容：



```
export JAVA_HOME=/usr/lib/jvm/java-1.8.0-openjdk-1.8.0.121-0.b13.el6_8.x86_64/jre
export PATH=$PATH:$JAVA_HOME
```

根据安装情况修改对应路径。

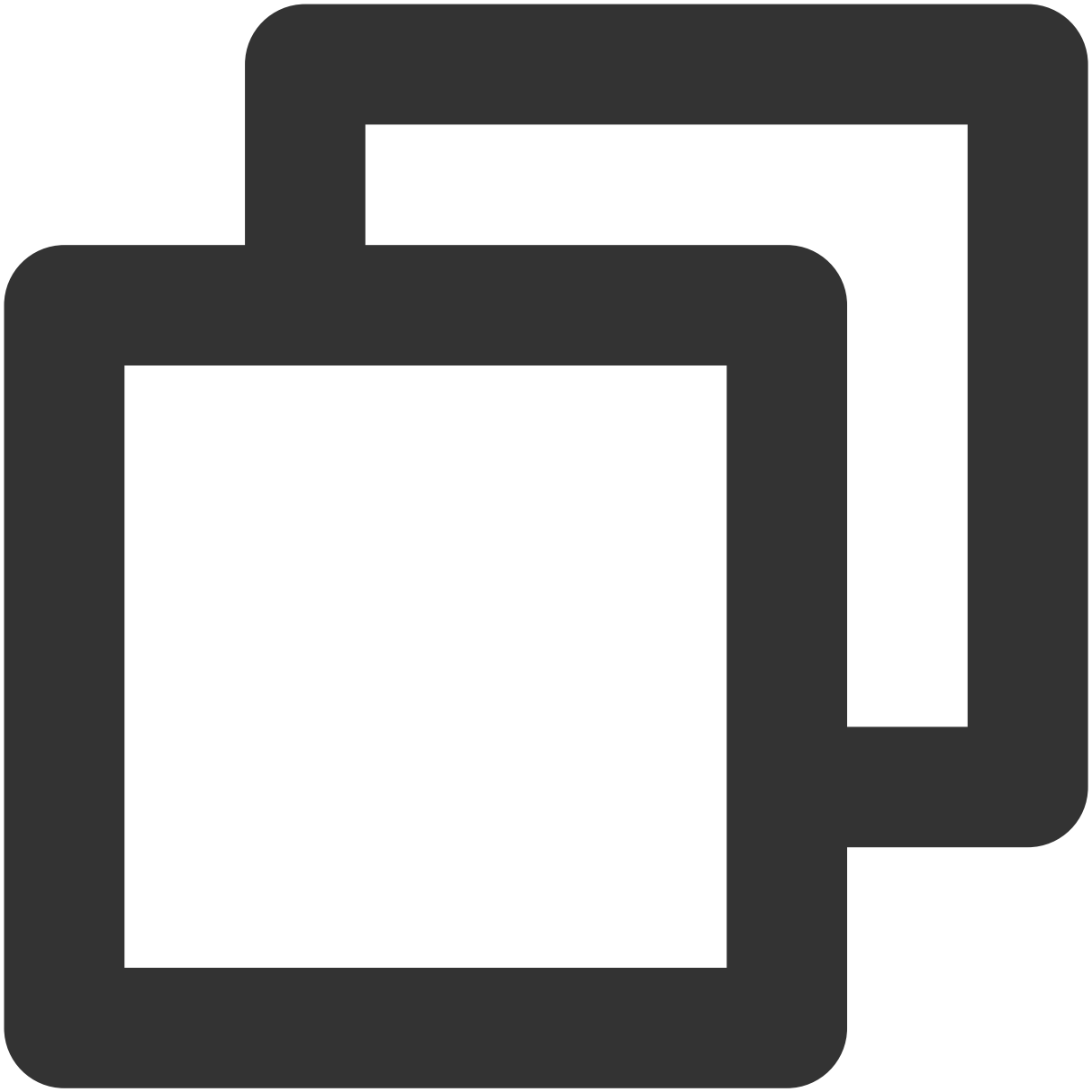
9. 解压 Hadoop，进入目录。



```
./bin/hadoop version
```

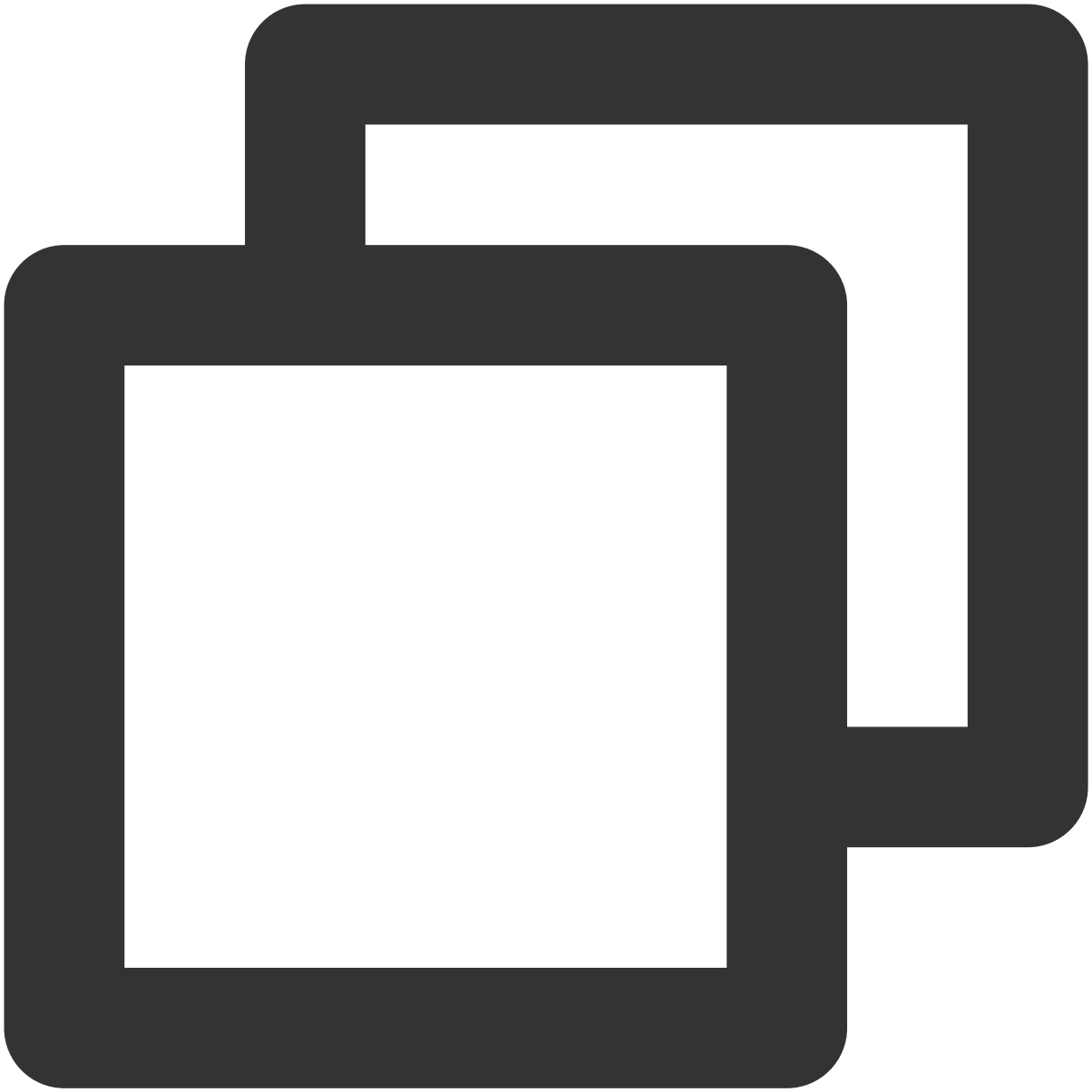
若能显示版本信息说明能正常运行。

10. 配置单机伪分布式（可根据需要搭建不同形式的集群）。



```
vim /etc/profile
```

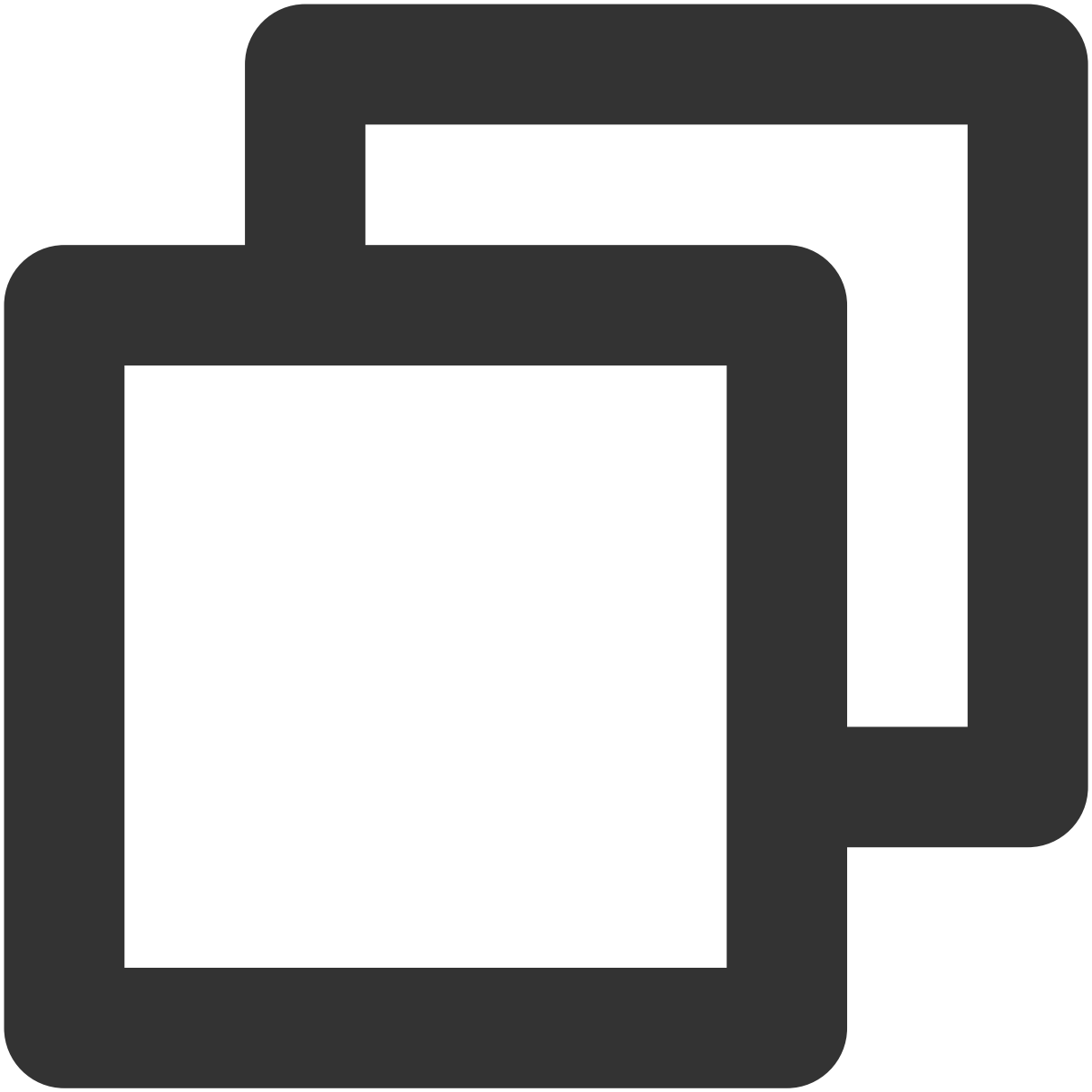
在文末加上下述内容：



```
export HADOOP_HOME=/usr/local/hadoop
export PATH=$HADOOP_HOME/bin:$PATH
```

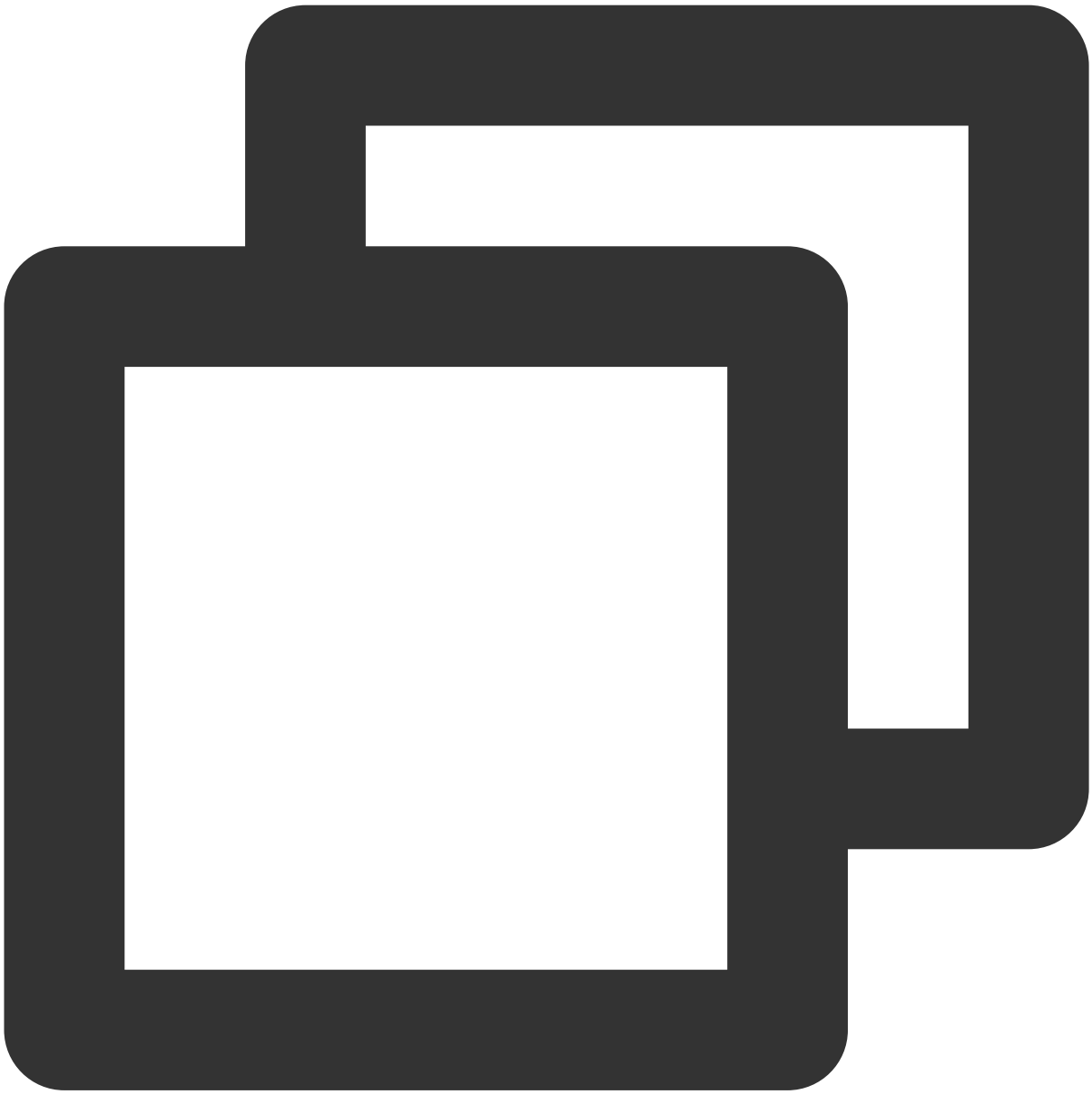
根据安装情况修改对应路径。

11. 修改 `/etc/hadoop/core-site.xml` 。



```
<configuration>
    <property>
        <name>hadoop.tmp.dir</name>
        <value>file:/usr/local/hadoop/tmp</value>
        <description>Abase for other temporary directories.</description>
    </property>
    <property>
        <name>fs.defaultFS</name>
        <value>hdfs://localhost:9000</value>
    </property>
</configuration>
```

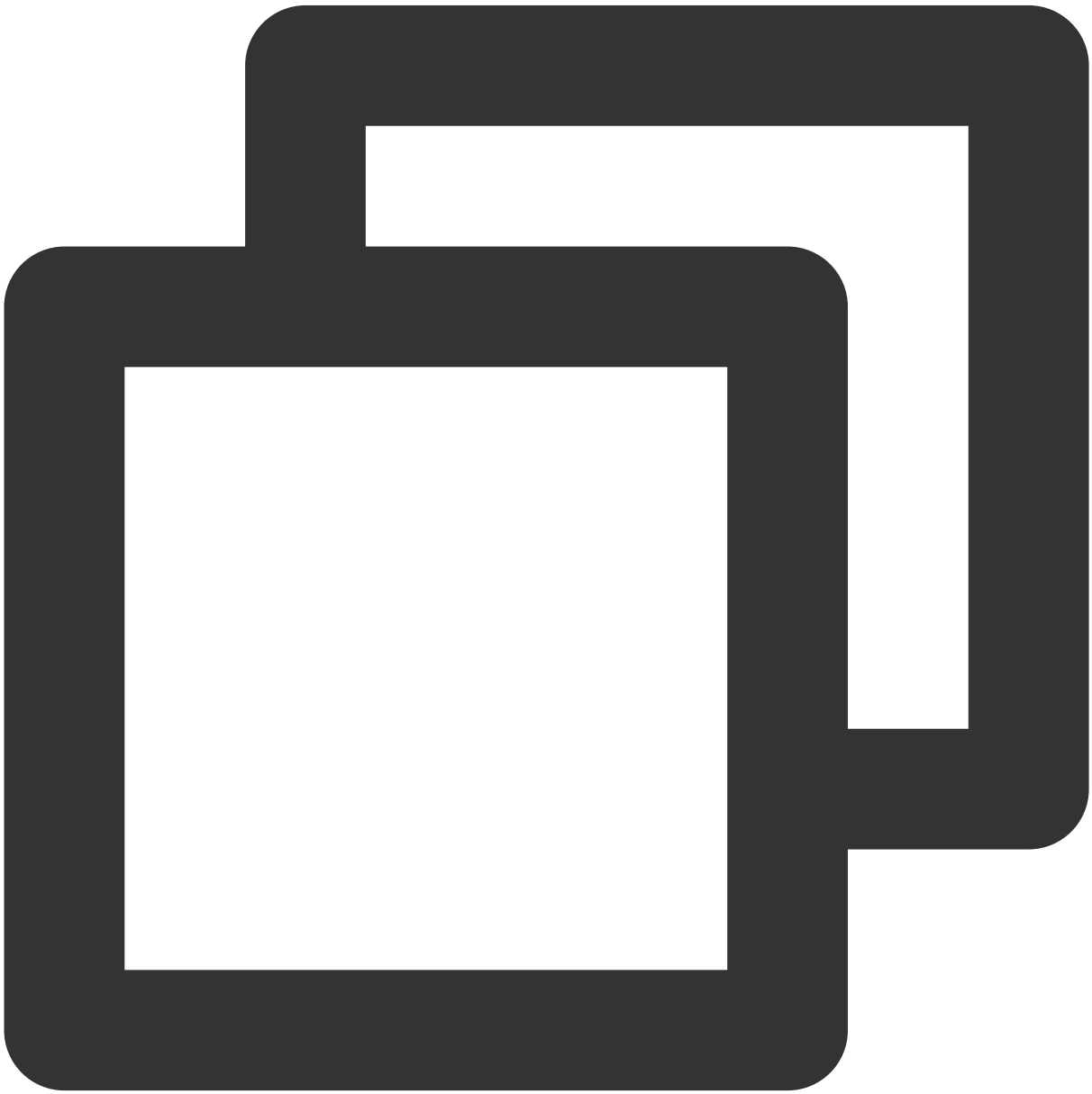
12. 修改 `/etc/hadoop/hdfs-site.xml` 。



```
<configuration>
    <property>
        <name>dfs.replication</name>
        <value>1</value>
    </property>
    <property>
        <name>dfs.namenode.name.dir</name>
```

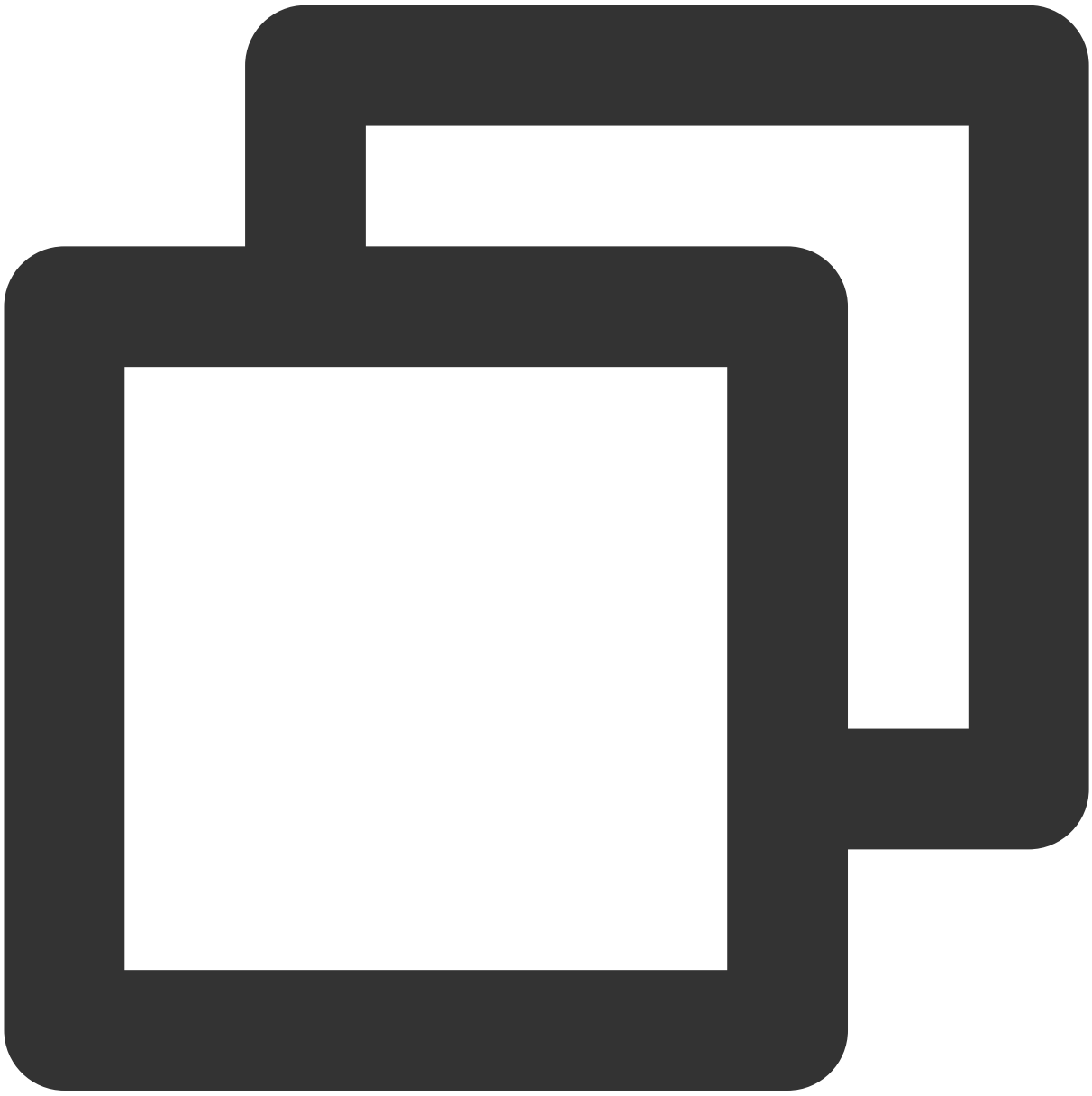
```
        <value>file:/usr/local/hadoop/tmp/dfs/name</value>
    </property>
    <property>
        <name>dfs.datanode.data.dir</name>
        <value>file:/usr/local/hadoop/tmp/dfs/data</value>
    </property>
</configuration>
```

13. 修改 `/etc/hadoop/hadoop-env.sh` 中的 JAVA_HOME 为 Java 的路径。



```
export JAVA_HOME=/usr/lib/jvm/java-1.8.0-openjdk-1.8.0.121-0.b13.el6_8.x86_64/jre
```

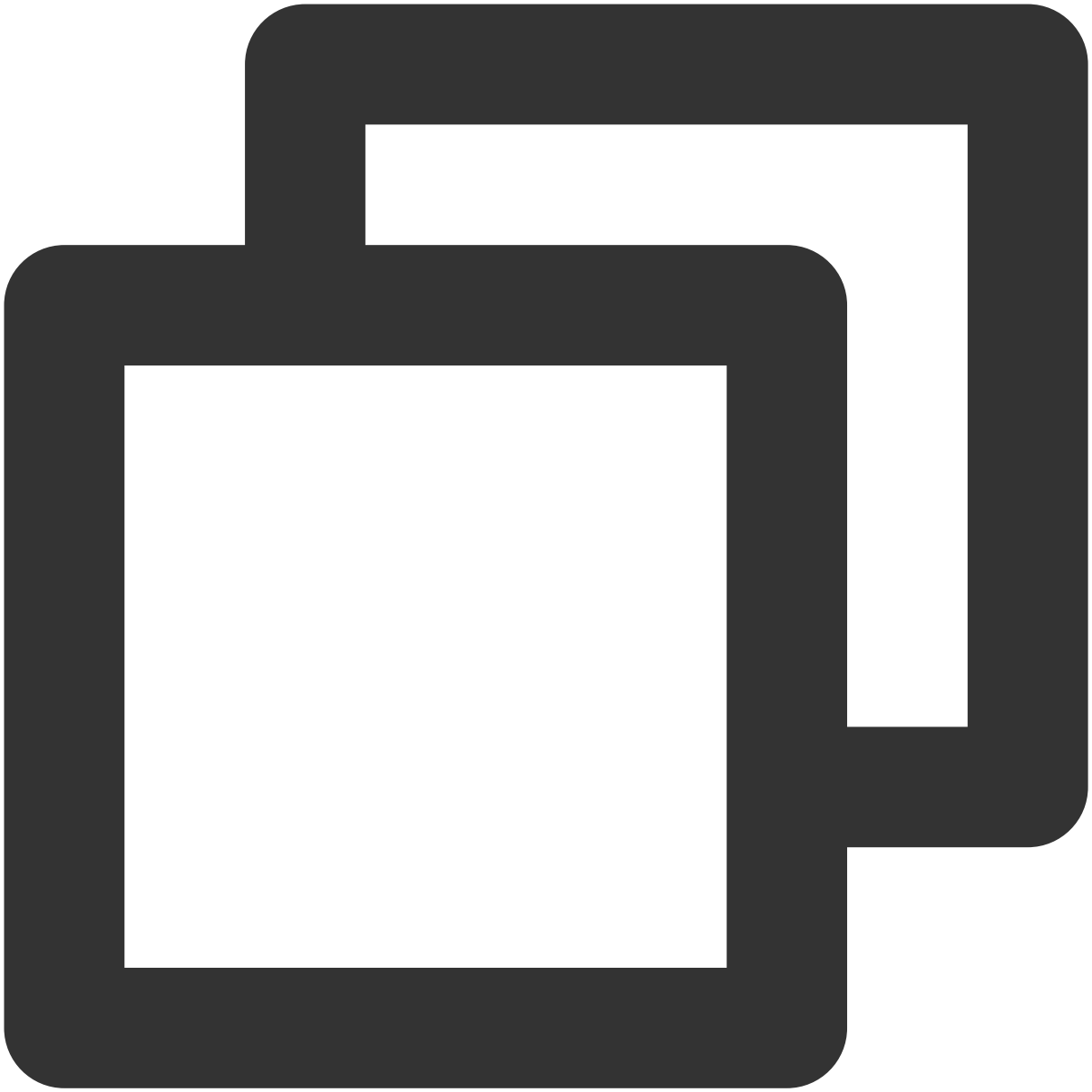
14. 执行 NameNode 格式化。



```
./bin/hdfs namenode -format
```

显示 `Exitting with status 0` 则表示成功。

15. 启动 Hadoop。



```
./sbin/start-dfs.sh
```

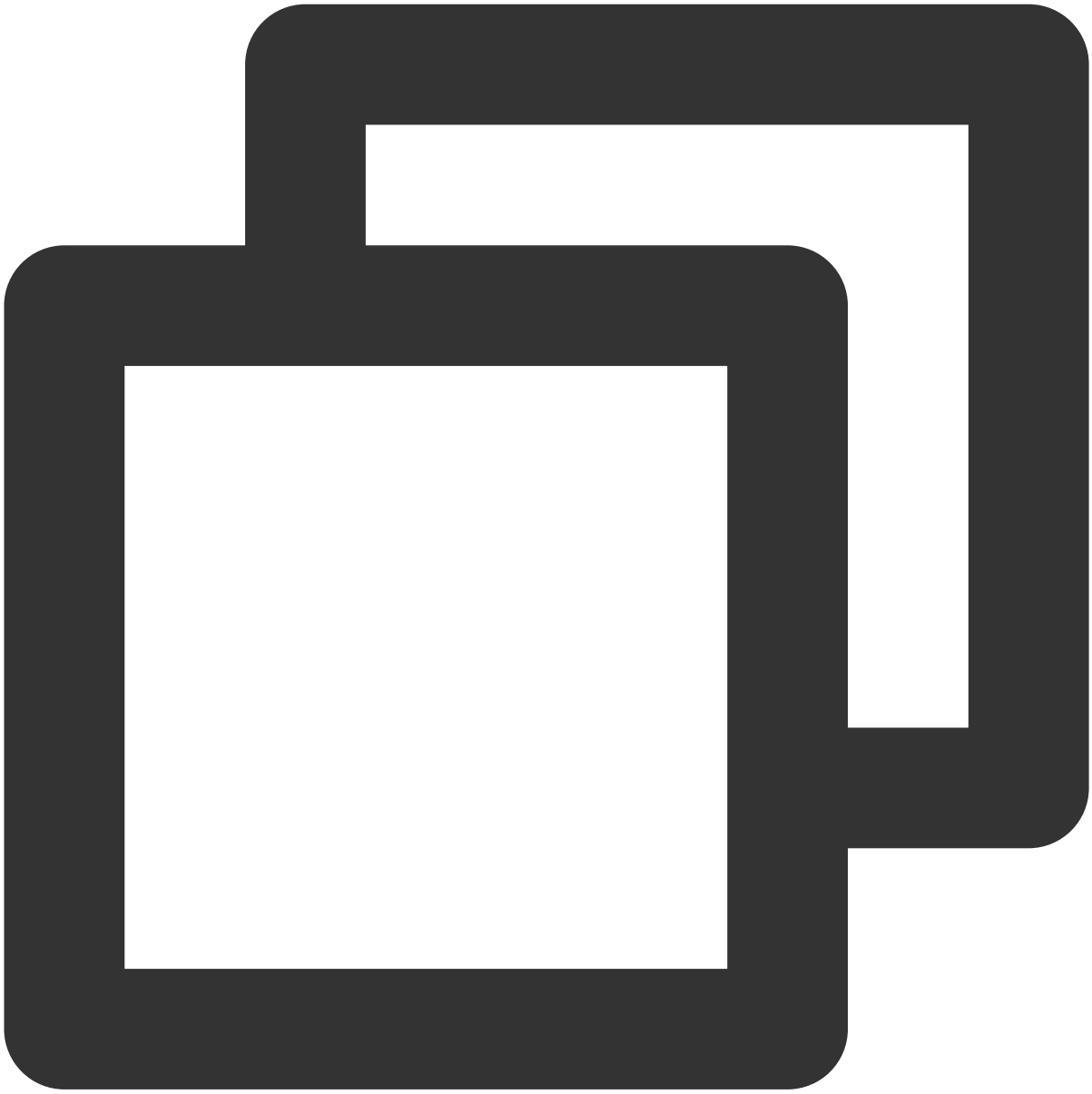
成功启动会存在 `NameNode` 进程，`DataNode` 进程，`SecondaryNameNode` 进程。

安装 Spark

访问 Spark 官网 下载所需要的版本。
因为之前安装了 Hadoop，所以选择使用 `Pre-build with user-provided Apache Hadoop` 。
说明：

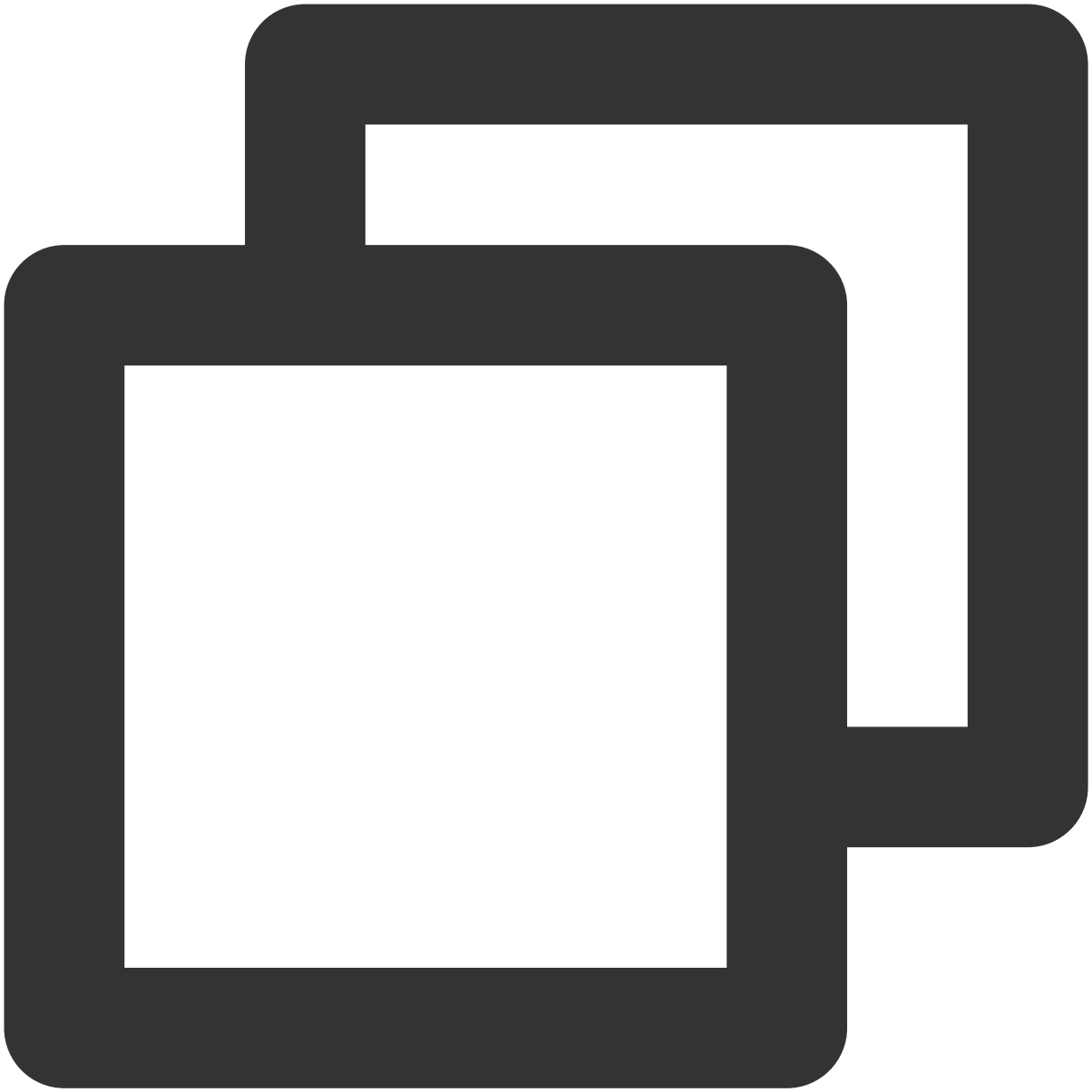
本示例同样使用 `hadoop` 用户进行操作。

1. 解压进入目录。
2. 修改配置文件。



```
cp ./conf/spark-env.sh.template ./conf/spark-env.sh
vim ./conf/spark-env.sh
```

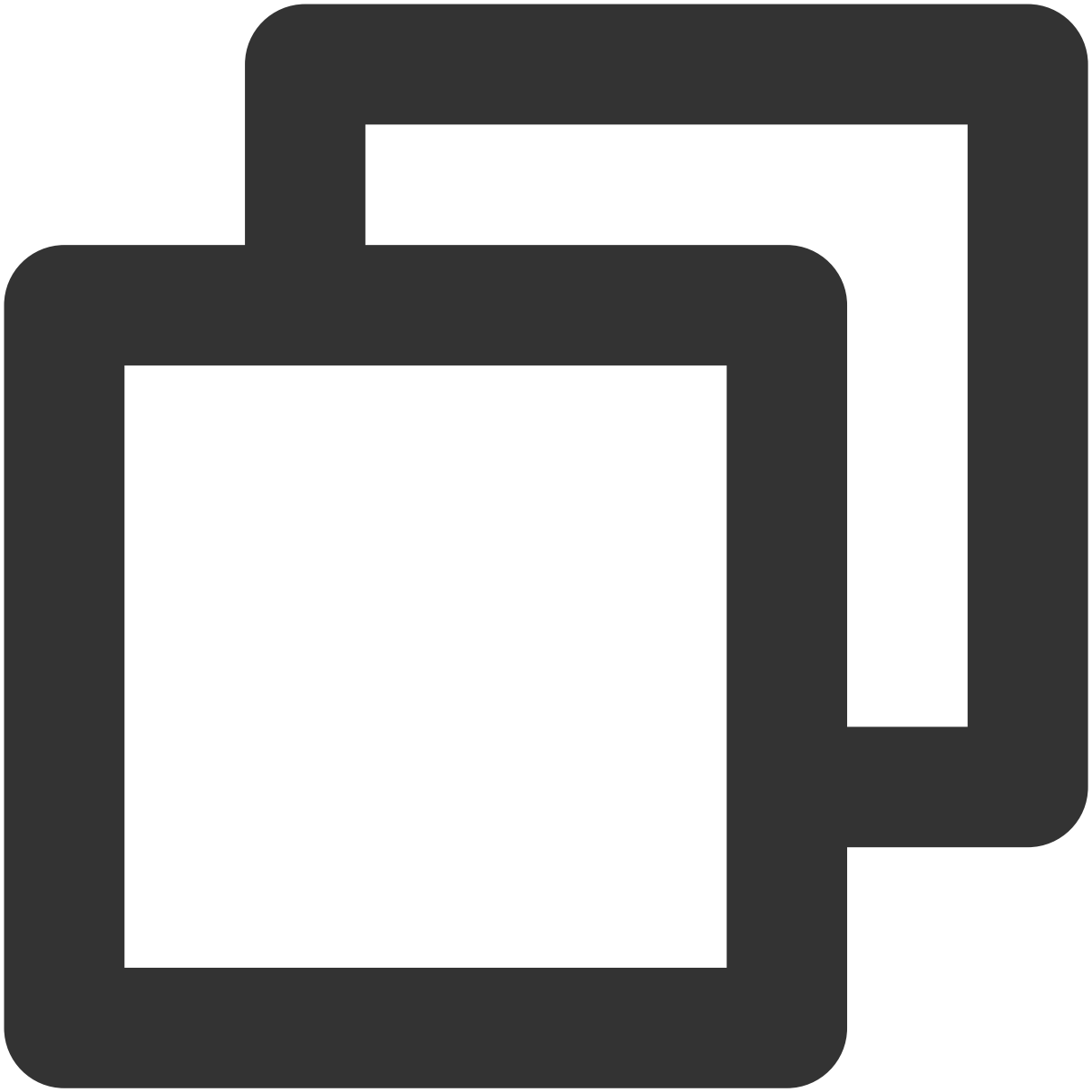
在第一行添加下述内容：



```
export SPARK_DIST_CLASSPATH=$(/usr/local/hadoop/bin/hadoop classpath)
```

根据 hadoop 安装情况修改路径。

3. 运行示例。



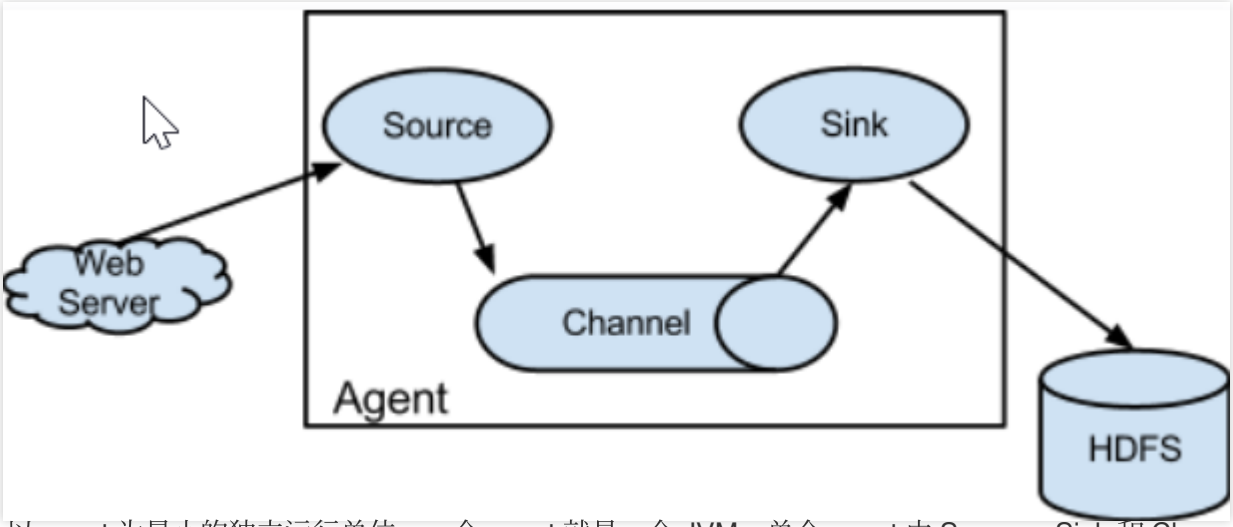
```
bin/run-example SparkPi
```

若成功安装可以看到程序输出 π 的近似值。

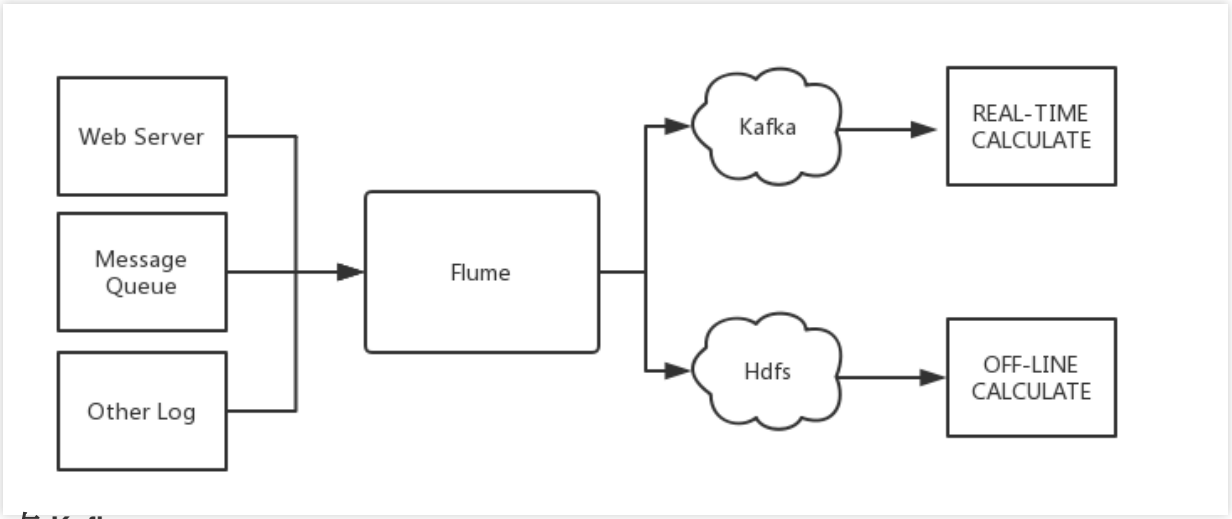
Flume 接入 CKafka

最近更新时间：2024-05-31 12:08:45

Apache Flume 是一个分布式、可靠、高可用的日志收集系统，支持各种各样的数据来源（例如 HTTP、Log 文件、JMS、监听端口数据等），能将这些数据源的海量日志数据进行高效收集、聚合、移动，最后存储到指定存储系统中（例如 Kafka、分布式文件系统、Solr 搜索服务器等）。
Flume 基本结构如下：



Flume 以 agent 为最小的独立运行单位。一个 agent 就是一个 JVM，单个 agent 由 Source、Sink 和 Channel 三大组件构成。



Flume 与 Kafka

把数据存储到 HDFS 或者 HBase 等下游存储模块或者计算模块时需要考虑各种复杂的场景，例如并发写入的量以及系统承载压力、网络延迟等问题。Flume 作为灵活的分布式系统具有多种接口，同时提供可定制化的管道。
在生产处理环节中，当生产与处理速度不一致时，Kafka 可以充当缓存角色。Kafka 拥有 partition 结构以及采用 append 追加数据，使 Kafka 具有优秀的吞吐能力；同时其拥有 replication 结构，使 Kafka 具有很高的容错性。

所以将 Flume 和 Kafka 结合起来，可以满足生产环境中绝大多数要求。

Flume 接入开源 Kafka

准备工作

下载 Apache Flume （1.6.0以上版本兼容 Kafka）

下载 Kafka工具包 （0.9.x以上版本，0.8已经不支持）

确认 Kafka 的 Source、 Sink 组件已经在 Flume 中。

接入方式

Kafka 可作为 Source 或者 Sink 端对消息进行导入或者导出。

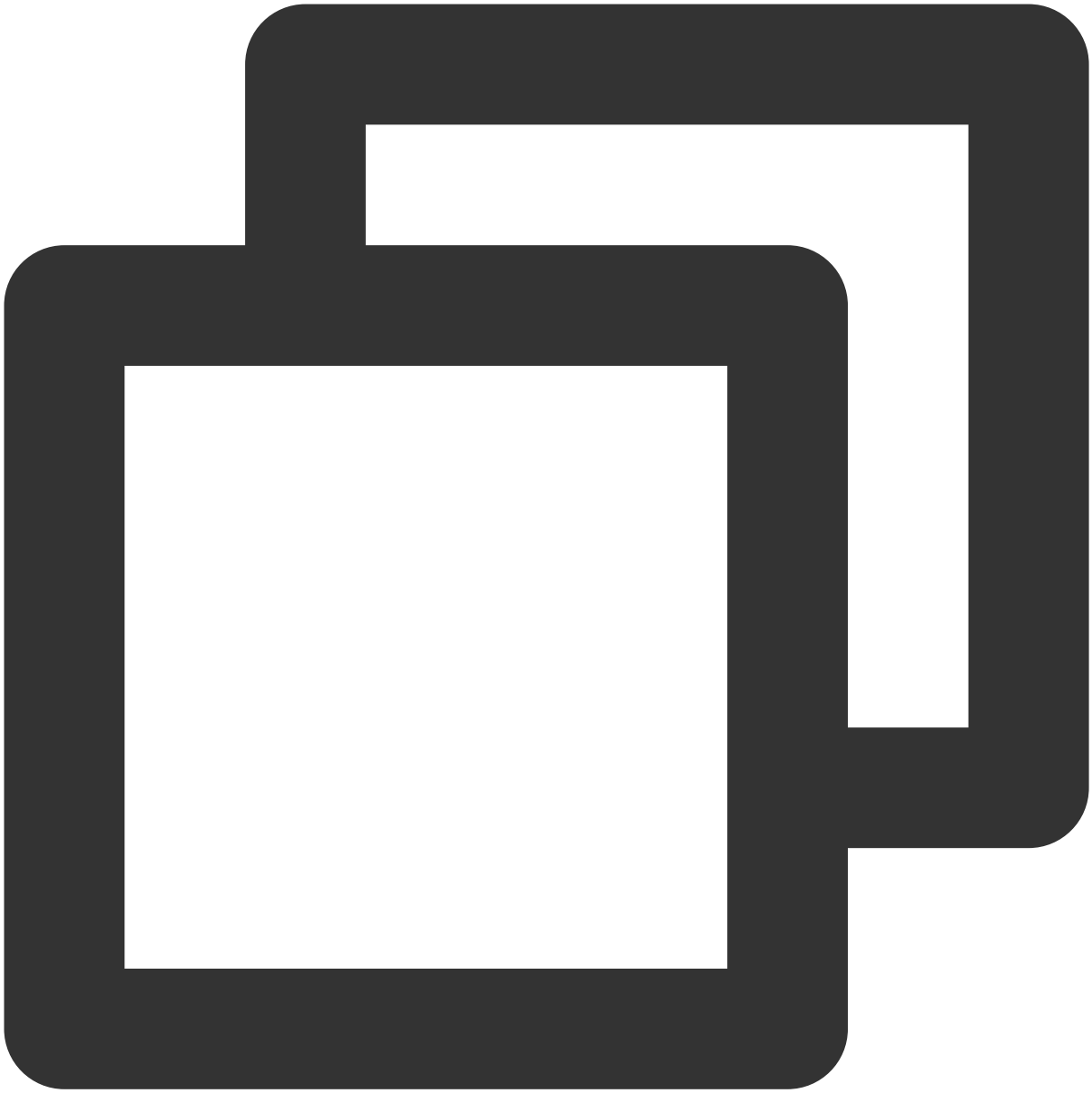
Kafka Source

Kafka Sink

配置 kafka 作为消息来源，即将自己作为消费者，从 Kafka 中拉取数据传入到指定 Sink 中。主要配置选项如下：

配置项	说明
channels	自己配置的 Channel
type	必须为：org.apache.flume.source.kafka.KafkaSource
kafka.bootstrap.servers	Kafka Broker 的服务器地址
kafka.consumer.group.id	作为 Kafka 消费端的 Group ID
kafka.topics	Kafka 中数据来源 Topic
batchSize	每次写入 Channel 的大小
batchDurationMillis	每次写入最大间隔时间

示例：



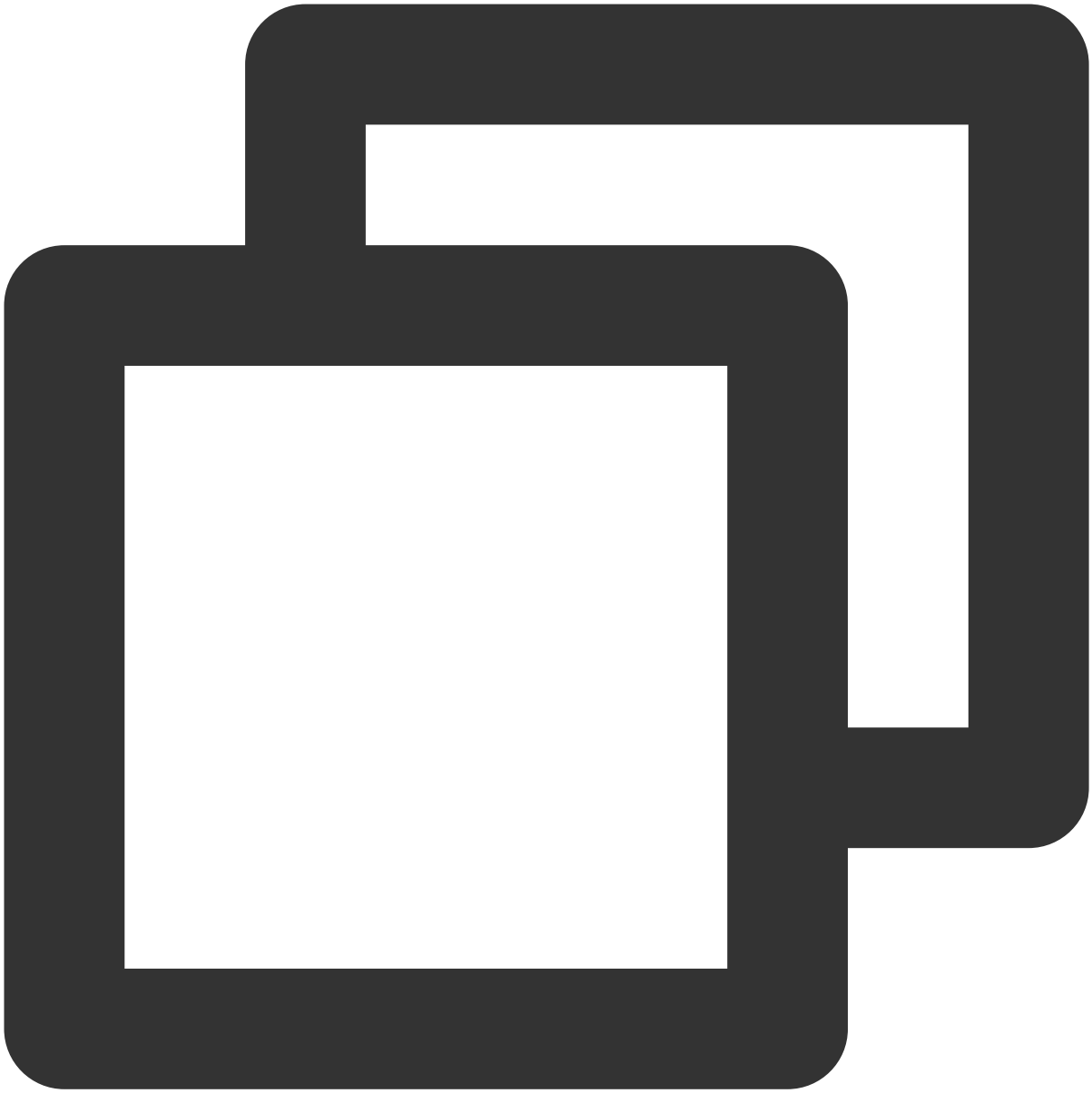
```
tier1.sources.source1.type = org.apache.flume.source.kafka.KafkaSource
tier1.sources.source1.channels = channel1
tier1.sources.source1.batchSize = 5000
tier1.sources.source1.batchDurationMillis = 2000
tier1.sources.source1.kafka.bootstrap.servers = localhost:9092
tier1.sources.source1.kafka.topics = test1, test2
tier1.sources.source1.kafka.consumer.group.id = custom.g.id
```

更多内容请参见 Apache Flume 官网。

配置 Kafka 作为内容接收方，即将自己作为生产者，推到 Kafka Server 中等待后续操作。主要配置选项如下：

配置项	说明
channel	自己配置的 Channel
type	必须为：org.apache.flume.sink.kafka.KafkaSink
kafka.bootstrap.servers	Kafka Broker 的服务器
kafka.topics	Kafka 中数据来源 Topic
kafka.flumeBatchSize	每次写入的 Bacth 大小
kafka.producer.acks	Kafka 生产者的生产策略

示例：



```
a1.sinks.k1.channel = c1
a1.sinks.k1.type = org.apache.flume.sink.kafka.KafkaSink
a1.sinks.k1.kafka.topic = mytopic
a1.sinks.k1.kafka.bootstrap.servers = localhost:9092
a1.sinks.k1.kafka.flumeBatchSize = 20
a1.sinks.k1.kafka.producer.acks = 1
```

更多内容请参见 Apache Flume 官网。

Flume 接入 CKafka

使用 CKafka 作为 Sink

使用 CKafka 作为 Source

步骤1：获取 CKafka 实例接入地址

1. 登录 CKafka 控制台。
2. 在左侧导航栏选择**实例列表**，单击实例的“ID”，进入实例基本信息页面。
3. 在实例的基本信息页面的**接入方式**模块，可获取实例的接入地址。

步骤2：创建 Topic

1. 在实例基本信息页面，选择顶部 **Topic 管理**页签。
2. 在 Topic 管理页面，单击**新建**，创建一个名为 flume_test 的 Topic。

步骤3：配置 Flume

1. 下载 Apache Flume 工具包并解压。
2. 编写配置文件 flume-kafka-sink.properties，以下是一个简单的 Java 语言 Demo（配置在解压目录的 conf 文件夹下），若无特殊要求则将自己的实例 IP 与 Topic 替换到配置文件当中即可。本例使用的 source 为 tail -F flume-test，即文件中新增的信息。

```
# Take kafka as a sink
agentckafka.sources = exectail
agentckafka.channels = memoryChannel
agentckafka.sinks = kafkaSink

# Set source type as needed
agentckafka.sources.exectail.type = exec
agentckafka.sources.exectail.command = tail -F ./flume-test
agentckafka.sources.exectail.batchSize=20
# Set source type as needed
agentckafka.sources.exectail.channels = memoryChannel

# Set the sink type. It is set to kafka here
agentckafka.sinks.kafkaSink.type= org.apache.flume.sink.kafka.KafkaSink
# Set the ip:port provided by Ckafka
agentckafka.sinks.kafkaSink.brokerList= 172.16.16.12:9092
# Set the topic that needs to import data. Create the topic in advance in the console before proceeding here
agentckafka.sinks.kafkaSink.topic= flume test
# Set sink channel
agentckafka.sinks.kafkaSink.channel = memoryChannel

# Each channel's type is defined.
agentckafka.channels.memoryChannel.type = memory
agentckafka.channels.memoryChannel.keep-alive = 10

# Other config values specific to each type of channel(sink or source)
# can be defined as well
# In this case, it specifies the capacity of the memory channel
agentckafka.channels.memoryChannel.capacity = 1000
agentckafka.channels.memoryChannel.transactionCapacity =1000
```

If you have a special source, you can configure it by yourself. The simplest example is used here

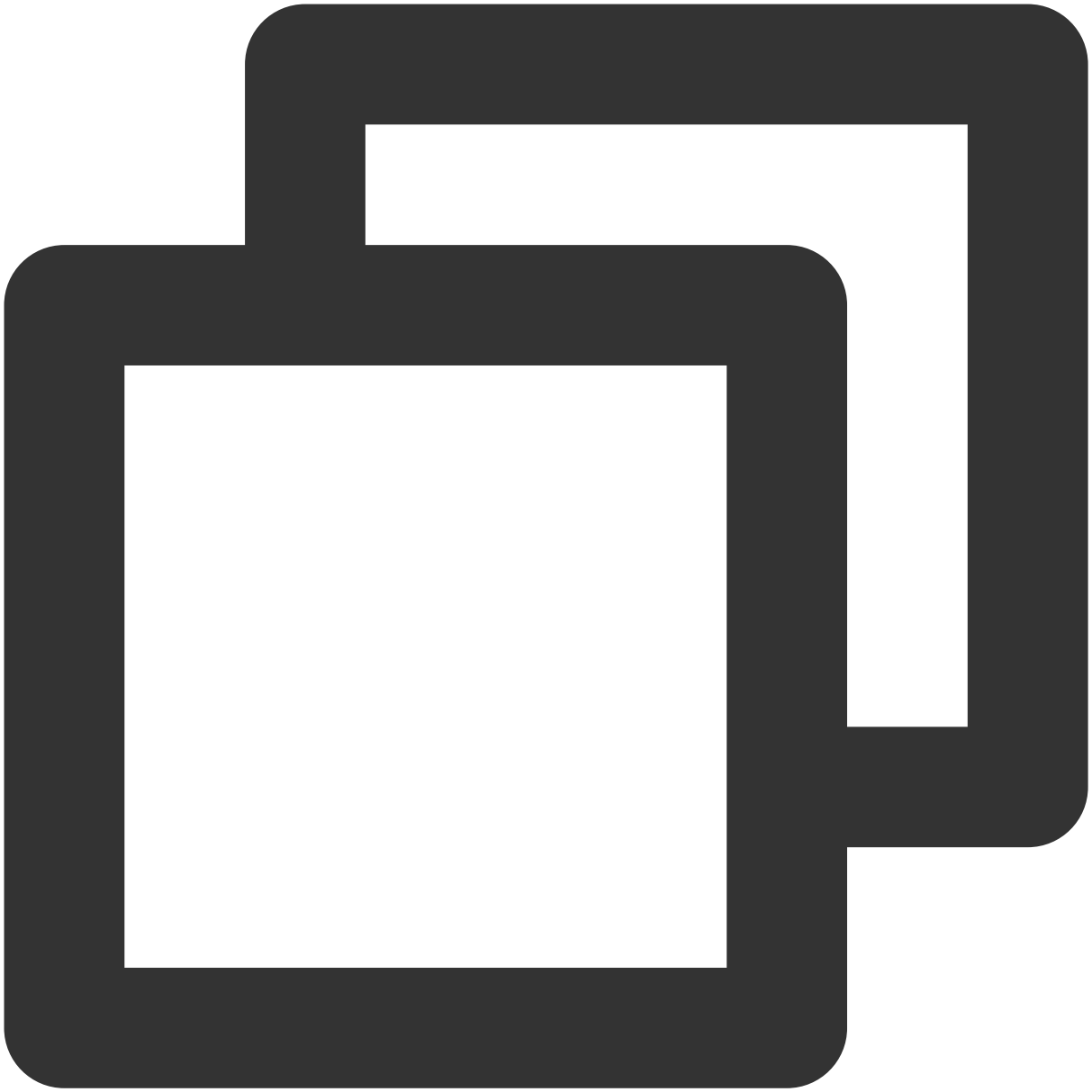
Configure instance IP

Configure topic

Configuration for using CKafka as the sink

Configuration for using CKafka as the sink

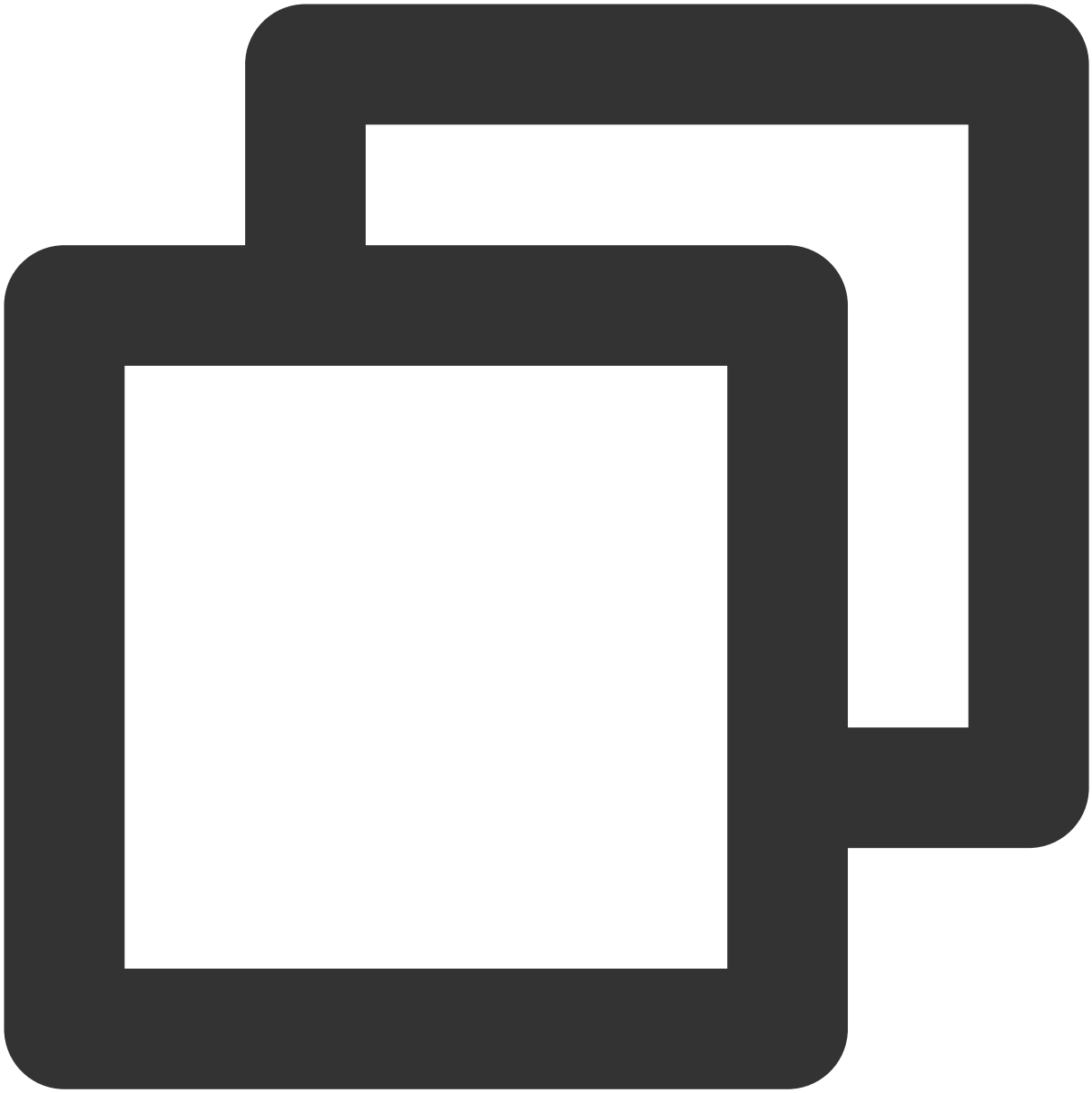
代码示例如下：



```
# 以kafka作为sink的demo
agentckafka.source = exectail
agentckafka.channels = memoryChannel
agentckafka.sinks = kafkaSink

# 设置source类型，根据不同需求而设置。若有特殊source可自行配置，此处使用最简单的例子
agentckafka.sources.exectail.type = exec
agentckafka.sources.exetail.command = tail -F ./flume.test
agentckafka.sources.exectail.batchSize = 20
# 设置source channel
agentckafka.sources.exectail.channels = memoryChannel
```

```
# 设置sink类型，此处设置为kafka
agentckafka.sinks.kafkaSink.type = org.apache.flume.sink.kafka.KafkaSink
# 此处设置ckafka提供的ip:port
agentckafka.sinks.kafkaSink.brokerList = 172.16.16.12:9092 # 配置实例IP
# 此处设置需要导入数据的topic，请先在控制台提前创建好topic
agentckafka.sinks.kafkaSink.topic = flume test #配置topic
# 设置sink channel
agentckafka.sinks.kafkaSink.channel = memoryChannel

# Channel使用默认配置
# Each channel's type is defined.
agentckafka.channels.memoryChannel.type = memory
agentckafka.channels.memoryChannel.keep-alive = 10

# Other config values specific to each type of channel(sink or source) can be defin
# In this case, it specifies the capacity of the memory channel
agentckafka.channels.memoryChannel.capacity = 1000
agentckafka.channels.memoryChannel.transactionCapacity = 1000
```

3. 执行如下命令启动 Flume。

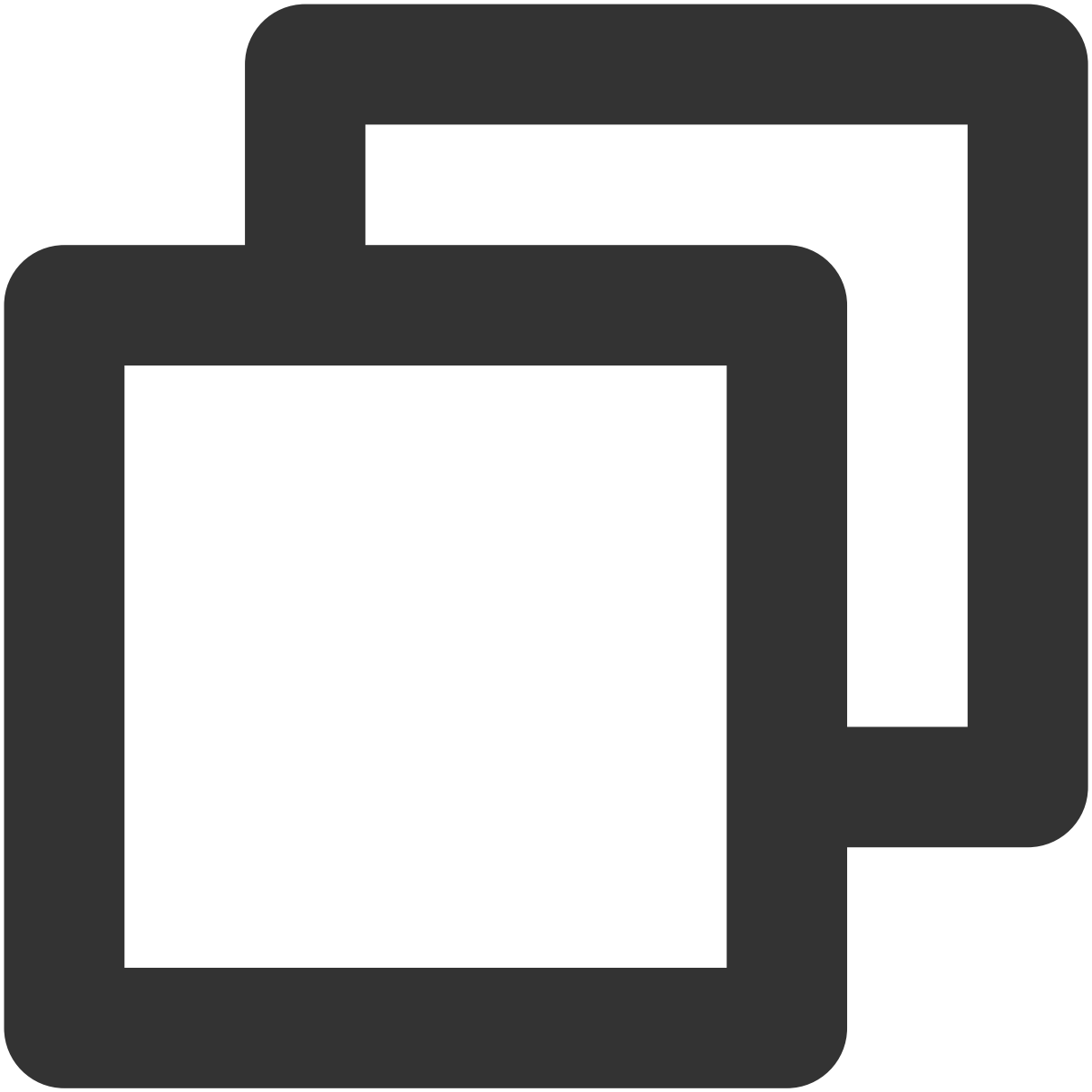


```
./bin/flume-ng agent -n agentckafka -c conf -f conf/flume-kafka-sink.properties
```

4. 写入消息到 flume-test 文件中，此时消息将由 Flume 写入到 CKafka。

```
[root@VM_16_17_centos apache-flume-1.7.0-bin]# cat flume-test
ckafka
```

5. 启动 CKafka 客户端进行消费。

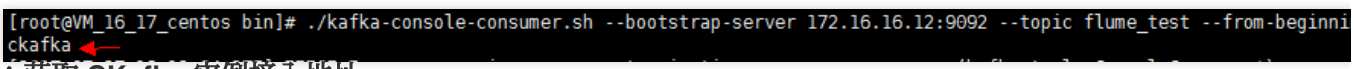


```
./kafka-console-consumer.sh --bootstrap-server xx.xx.xx.xx:xxxx --topic flume_test
```

说明：

bootstrap-server 填写刚创建的 CKafka 实例的接入地址，topic 填写刚创建的 Topic 名称。

可以看到刚才的消息被消费出来。



步骤1：获取 CKafka 实例接入地址

1. 登录 CKafka 控制台。

2. 在左侧导航栏选择**实例列表**，单击实例的“ID”，进入实例基本信息页面。
3. 在实例的基本信息页面的**接入方式**模块，可获取实例的接入地址。

步骤2：创建 Topic

1. 在实例基本信息页面，选择顶部 **Topic 管理**页签。
2. 在 Topic 管理页面，单击**新建**，创建一个名为 flume_test 的 Topic。

步骤3：配置 Flume

1. 下载 Apache Flume 工具包并解压 。
2. 编写配置文件 flume-kafka-source.properties，以下是一个简单的 Demo（配置在解压目录的 conf 文件夹下）。若无特殊要求则将自己的实例 IP 与 Topic 替换到配置文件当中即可。此处使用的 sink 为 logger。

```
# Take kafka as a source
agentckafka.sources = kafkaSource
agentckafka.channels = memoryChannel
agentckafka.sinks = loggerSink

# Set the source type. It is set to kafka here
agentckafka.sources.kafkaSource.type= org.apache.flume.source.kafka.KafkaSource
# Set the ip:port provided by Ckafka
agentckafka.sources.kafkaSource.kafka.bootstrap.servers= 172.16.16.12:9092
# Set the topic that needs to export data. Create the topic in advance in the console before proceeding here
agentckafka.sources.kafkaSource.kafka.topics= flume_test
# Set a processing method for the case when the offset data is not found
agentckafka.sources.kafkaSource.kafka.consumer.auto.offset.reset= earliest
# Set source channel
agentckafka.sources.kafkaSource.channels = memoryChannel

# Set sink
agentckafka.sinks.loggerSink.type = logger
# Set sink channel
agentckafka.sinks.loggerSink.channel = memoryChannel

# Each channel's type is defined.
agentckafka.channels.memoryChannel.type = memory
agentckafka.channels.memoryChannel.keep-alive = 10

# Other config values specific to each type of channel(sink or source)
# can be defined as well
# In this case, it specifies the capacity of the memory channel
agentckafka.channels.memoryChannel.capacity = 1000
agentckafka.channels.memoryChannel.transactionCapacity =1000
```

Instance IP

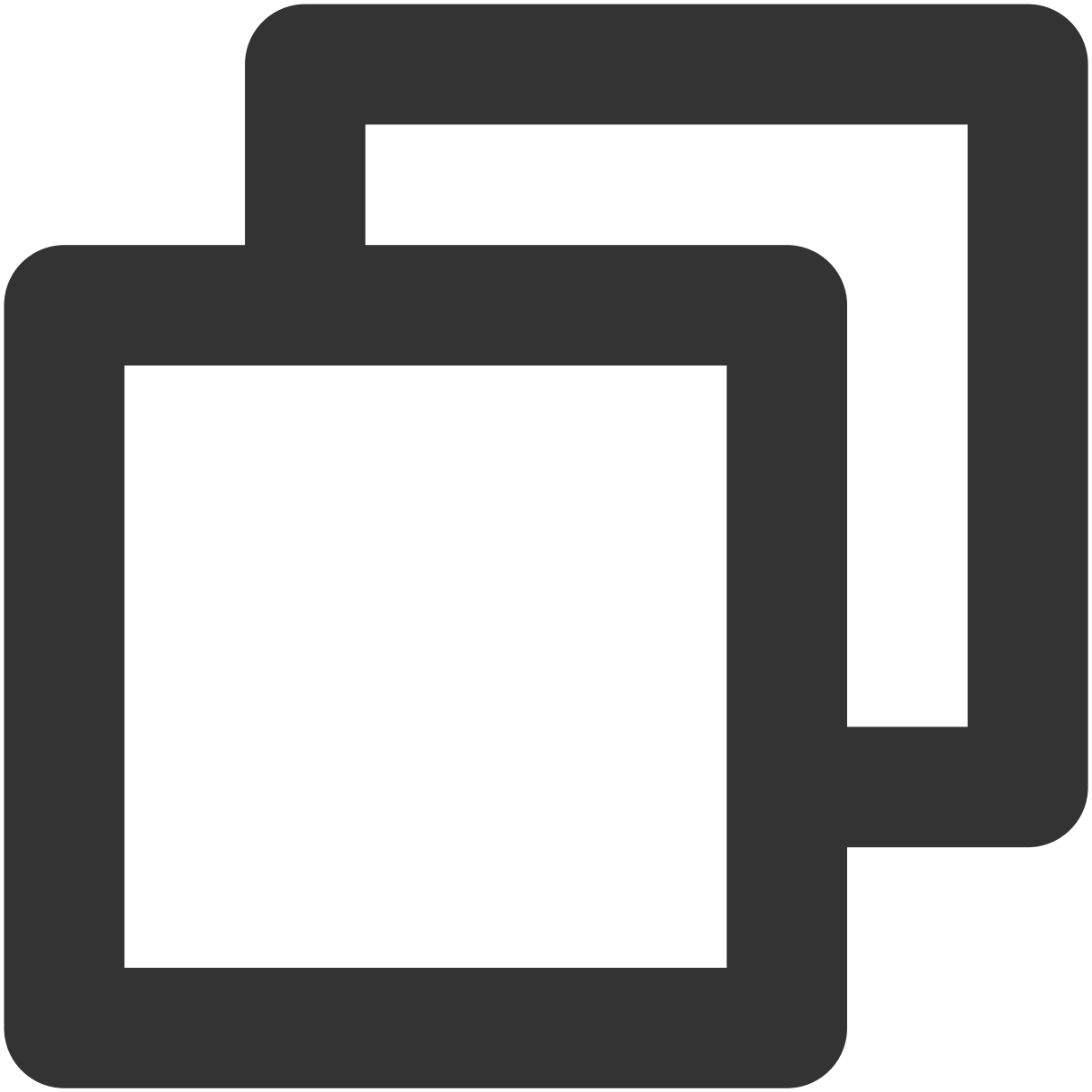
Source configuration

The topic just created

If you have a special sink, you can configure it by yourself

Use the default configuration for the channel

3. 执行如下命令启动 Flume。



```
./bin/flume-ng agent -n agentckafka -c conf -f conf/flume-kafka-source.properties
```

4. 查看 logger 输出信息（默认路径为 `logs/flume.log` ）。

```
Component type: SOURCE, name: kafkaSource started
- Event: { headers:{timestamp=1501136891423, topic=flume_test, partition=0} body: 63 6B 61 66 6B 61
```

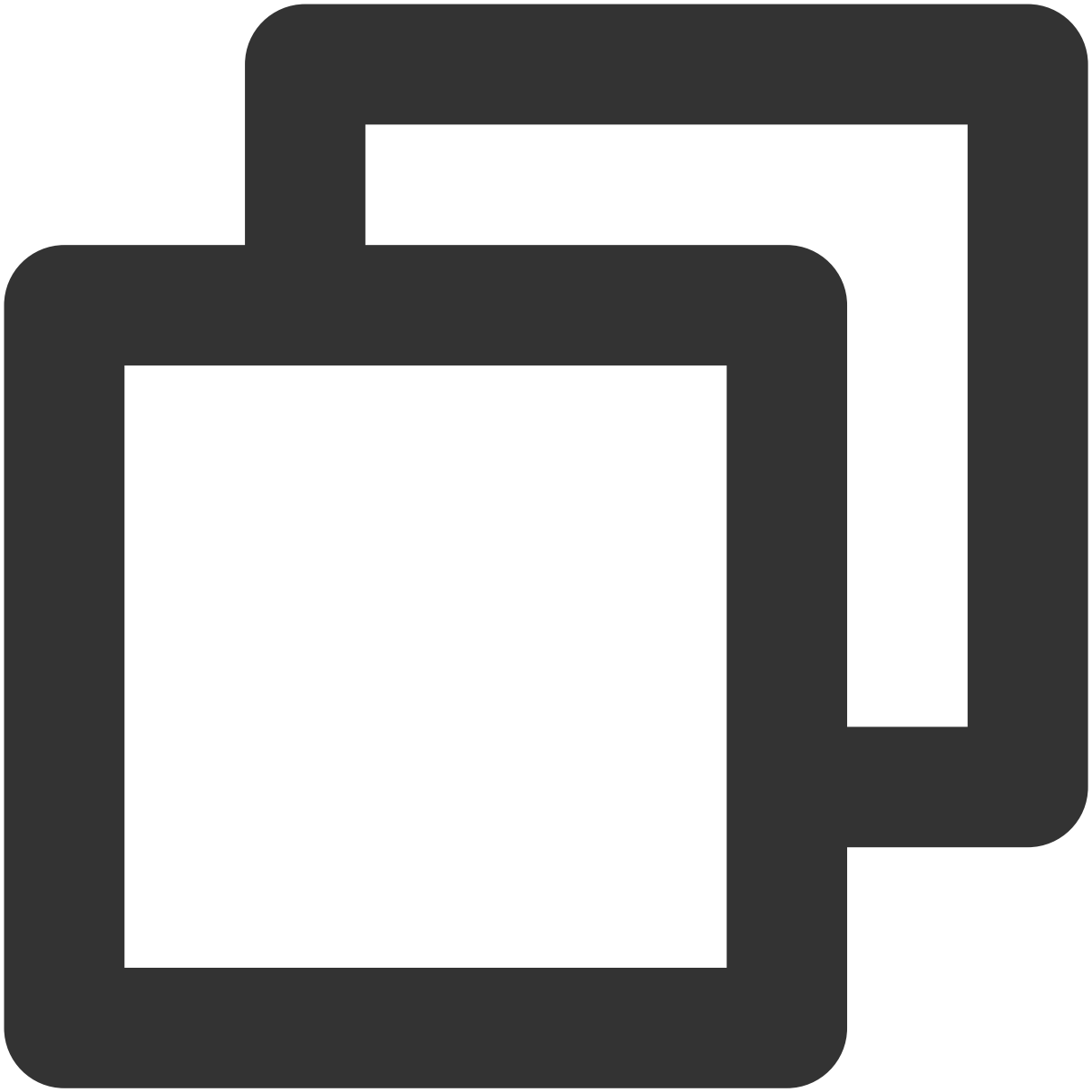
Kafka Connect 接入 CKafka

最近更新时间：2024-01-09 14:56:36

Kafka Connect 目前支持两种执行模式：standalone 和 distributed。

以 standalone 模式启动 connect

通过以下命令以 standalone 模式启动 connect：

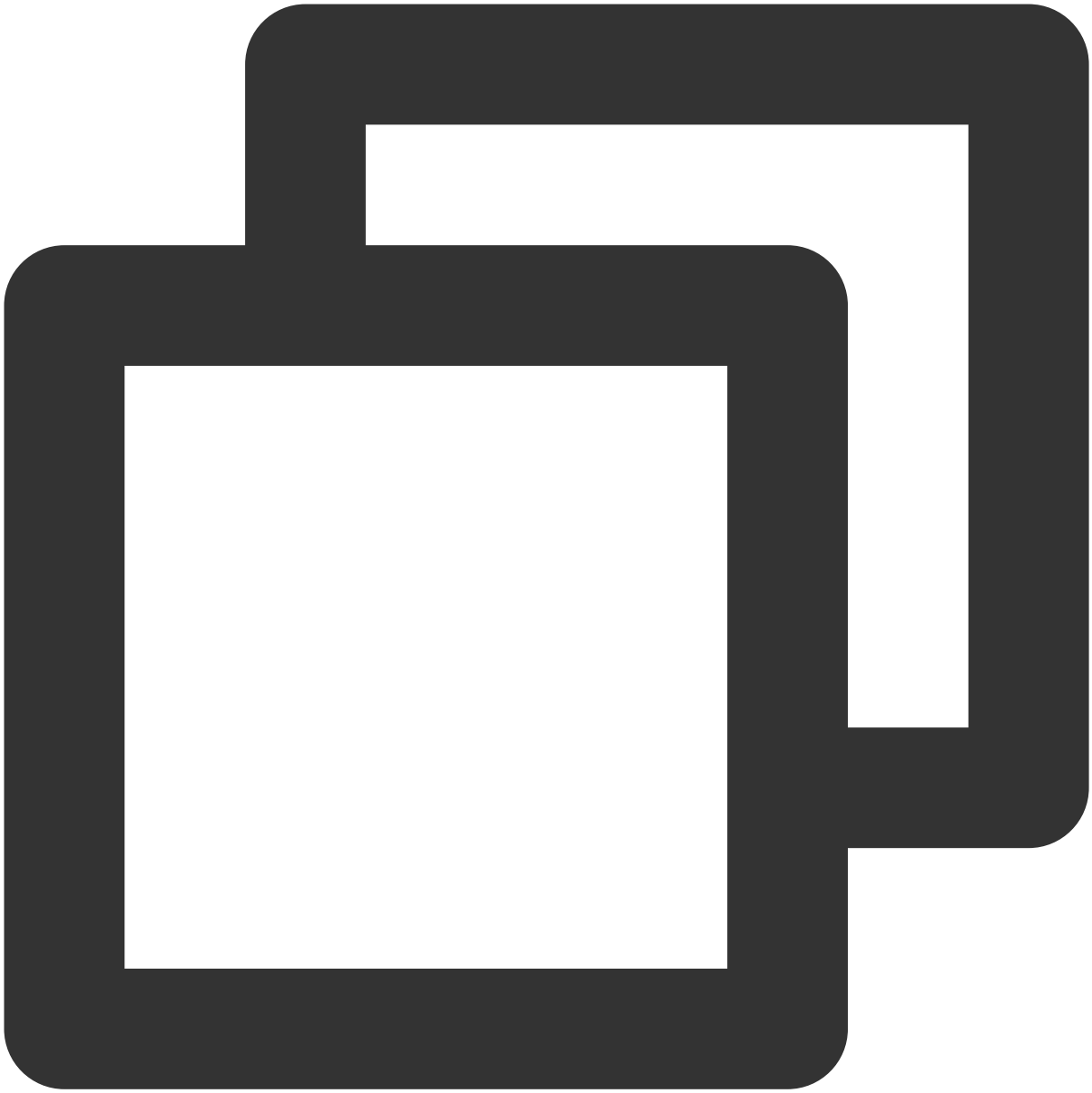


```
bin/connect-standalone.sh config/connect-standalone.properties connector1.propertie
```

接入 CKafka 与接入开源 Kafka 没有区别，仅需要修改 bootstrap.servers 为申请实例时分配的 IP。

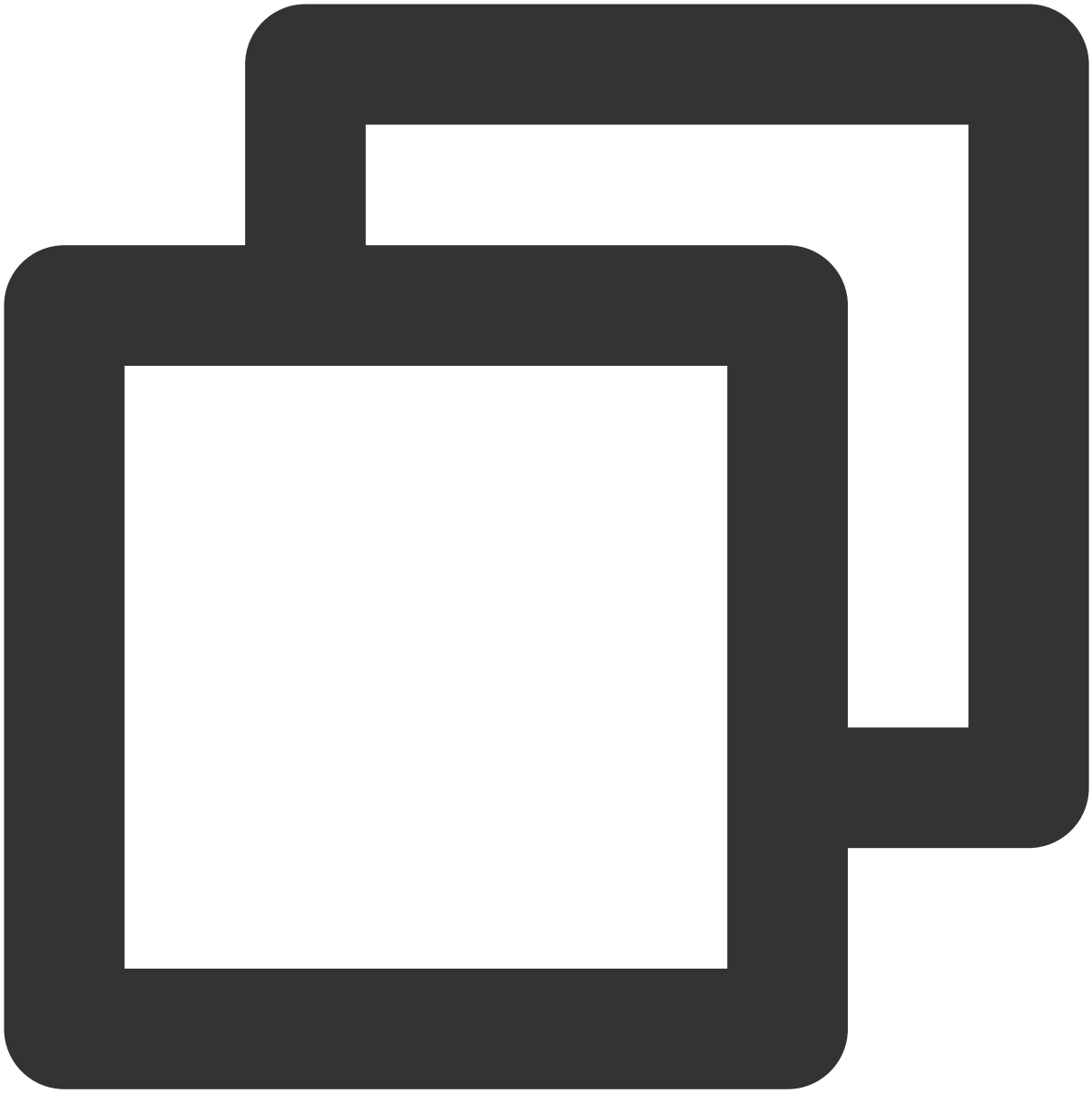
以 distributed 模式启动 connect

通过以下命令以 distributed 模式启动 connect：



```
bin/connect-distributed.sh config/connect-distributed.properties
```

该模式下，kafka connect 会将 offsets、configs 和 task status 信息存储在 kafka topic 中，存储的 topic 在 connect-distributed 中的以下字段配置：



```
config.storage.topic
offset.storage.topic
status.storage.topic
```

这三个 topic 需要手动创建，才能保证创建的属性符合 connect 的要求。

config.storage.topic 需要保证只有一个 partition，多副本且为 compact 模式。

offset.storage.topic 需要有多个 partition，多副本且为 compact 模式。

status.storage.topic 需要有多个 partition，多副本且为 compact 模式。

配置 bootstrap.servers 为申请实例时分配的 IP。

配置 group.id，用于标识 connect 集群，需要与消费者分组区分开来。

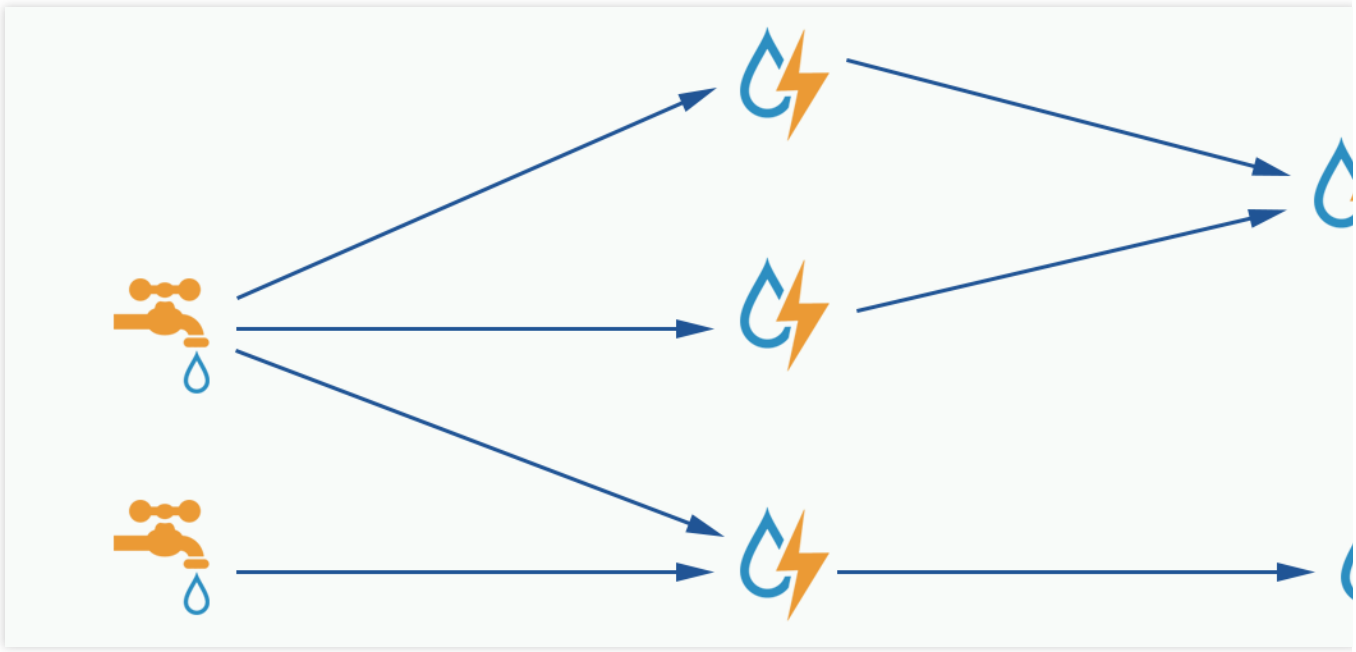
Storm 接入 CKafka

最近更新时间：2024-05-31 12:08:45

Storm 是一个分布式实时计算框架，能够对数据进行流式处理和提供通用性分布式 RPC 调用，可以实现处理事件亚秒级的延迟，适用于对延迟要求比较高的实时数据处理场景。

Storm 工作原理

在 Storm 的集群中有两种节点，控制节点 `Master Node` 和工作节点 `Worker Node` 。 `Master Node` 上运行 `Nimbus` 进程，用于资源分配与状态监控。 `Worker Node` 上运行 `Supervisor` 进程，监听工作任务，启动 `executor` 执行。整个 Storm 集群依赖 `zookeeper` 负责公共数据存放、集群状态监听、任务分配等功能。用户提交给 Storm 的数据处理程序称为 `topology` ，它处理的最小消息单位是 `tuple` ，一个任意对象的数组。 `topology` 由 `spout` 和 `bolt` 构成， `spout` 是产生 `tuple` 的源头， `bolt` 可以订阅任意 `spout` 或 `bolt` 发出的 `tuple` 进行处理。



Storm with CKafka

Storm 可以把 CKafka 作为 `spout` ，消费数据进行处理；也可以作为 `bolt` ，存放经过处理后的数据提供给其它组件消费。

测试环境

Centos6.8系统

package	version
maven	3.5.0
storm	2.1.0
ssh	5.3
Java	1.8

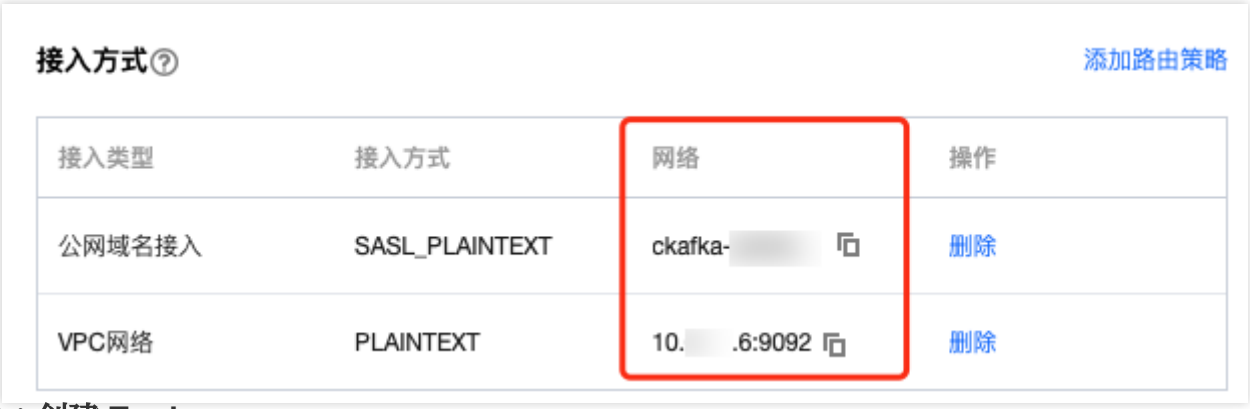
前提条件

下载并安装 JDK 8。具体操作，请参见 Download JDK 8。
下载并安装 Storm，参见 Apache Storm downloads。
已 创建 CKafka 实例。

操作步骤

步骤1：获取 CKafka 实例接入地址

1. 登录 CKafka 控制台。
2. 在左侧导航栏选择**实例列表**，单击实例的“ID”，进入实例基本信息页面。
3. 在实例的基本信息页面的**接入方式**模块，可获取实例的接入地址。

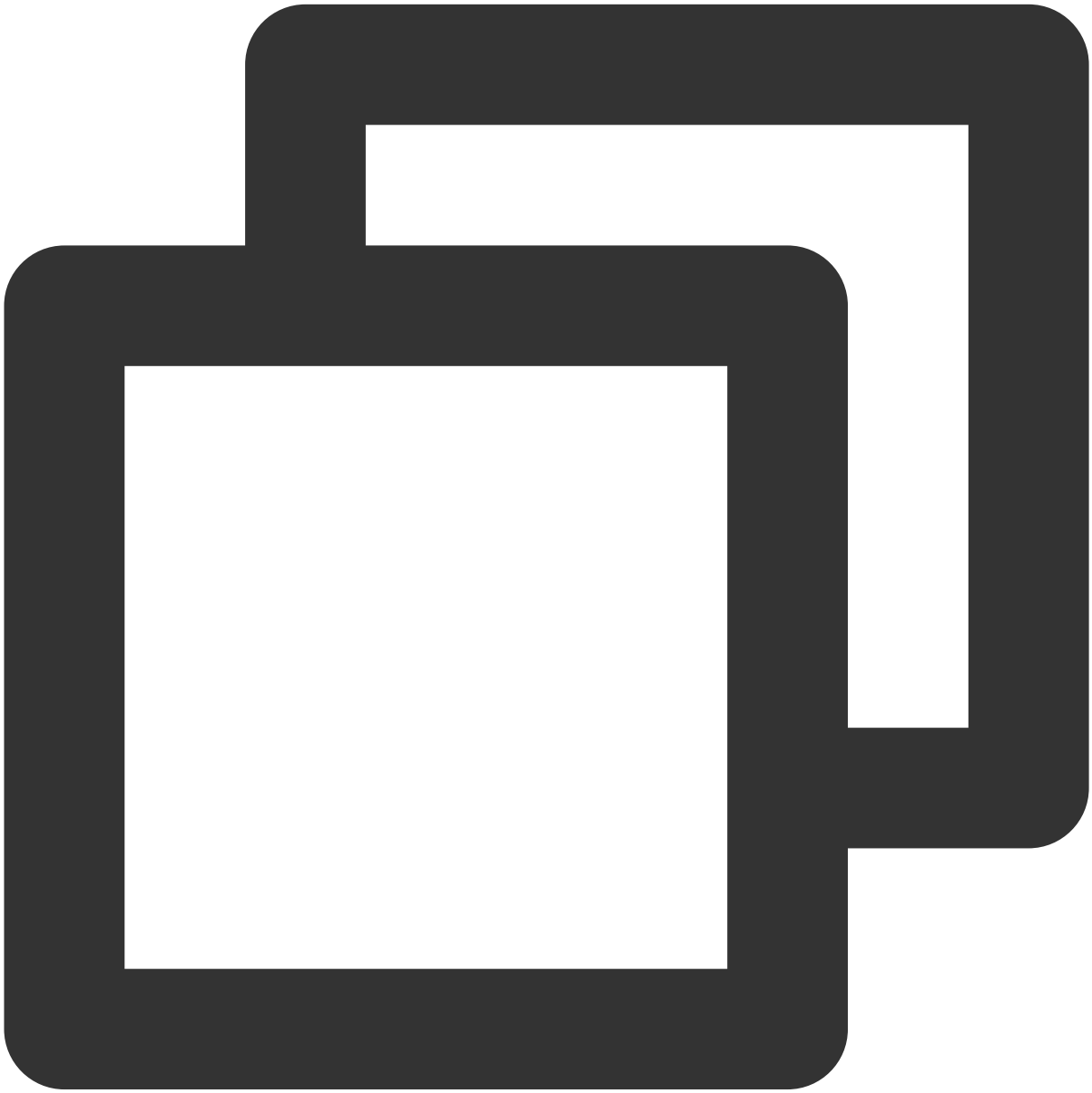


步骤2：创建 Topic

1. 在实例基本信息页面，选择顶部**Topic管理**页签。
2. 在 Topic 管理页面，单击**新建**，创建一个 Topic。

步骤3：添加 Maven 依赖

pom.xml 配置如下：



```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/200
  <modelVersion>4.0.0</modelVersion>
  <groupId>storm</groupId>
```

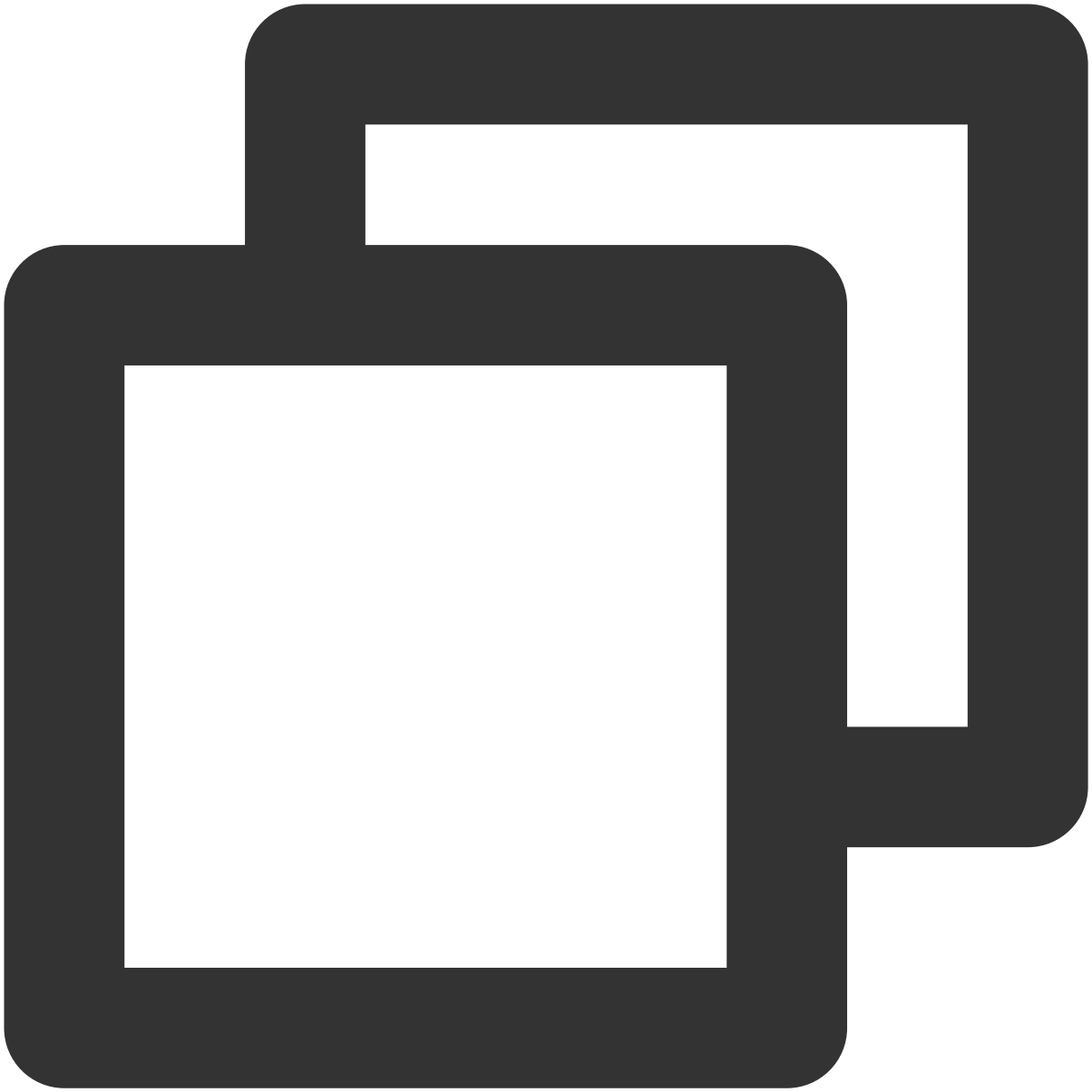
```
    <artifactId>storm</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>storm</name>
       <properties>
          <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
      </properties>
      <dependencies>
          <dependency>
              <groupId>org.apache.storm</groupId>
              <artifactId>storm-core</artifactId>
              <version>2.1.0</version>
          </dependency>
          <dependency>
              <groupId>org.apache.storm</groupId>
              <artifactId>storm-kafka-client</artifactId>
              <version>2.1.0</version>
          </dependency>
          <dependency>
              <groupId>org.apache.kafka</groupId>
              <artifactId>kafka_2.11</artifactId>
              <version>0.10.2.1</version>
              <exclusions>
                  <exclusion>
                      <groupId>org.slf4j</groupId>
                      <artifactId>slf4j-log4j12</artifactId>
                  </exclusion>
              </exclusions>
          </dependency>
          <dependency>
              <groupId>junit</groupId>
              <artifactId>junit</artifactId>
              <version>4.12</version>
              <scope>test</scope>
          </dependency>
      </dependencies>

      <build>
          <plugins>
              <plugin>
                  <artifactId>maven-assembly-plugin</artifactId>
                  <configuration>
                      <descriptorRefs>
                          <descriptorRef>jar-with-dependencies</descriptorRef>
                      </descriptorRefs>
                      <archive>
                          <manifest>
                              <mainClass>ExclamationTopology</mainClass>
```

```
                    </manifest>
                </archive>
            </configuration>
            <executions>
                <execution>
                    <id>make-assembly</id>
                    <phase>package</phase>
                    <goals>
                        <goal>single</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <configuration>
                <source>1.8</source>
                <target>1.8</target>
            </configuration>
        </plugin>
    </plugins>
</build>
</project>
```

步骤4：生产消息

使用 spout/bolt

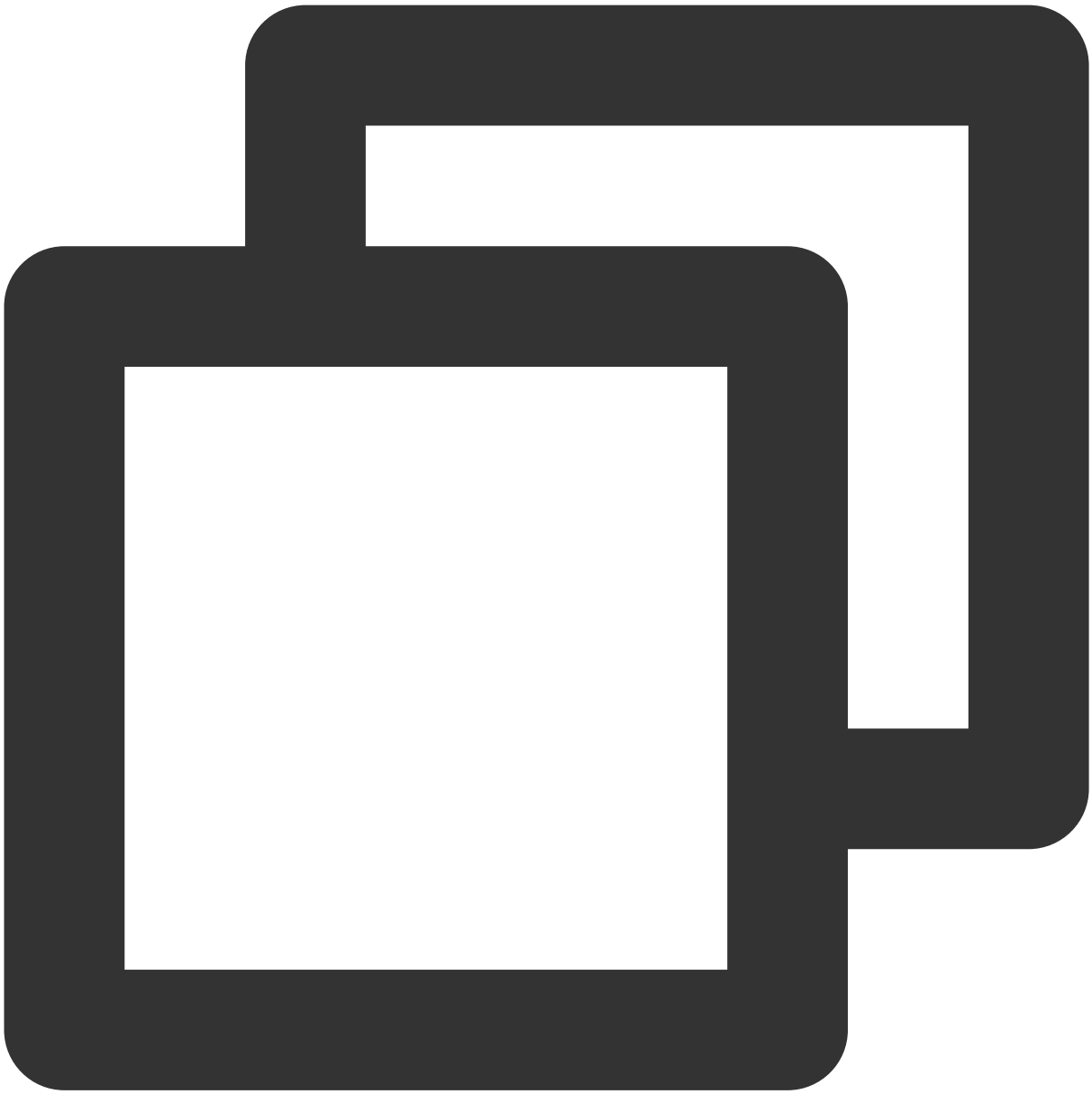
topology 代码：



```
//TopologyKafkaProducerSpout.java
import org.apache.storm.Config;
import org.apache.storm.LocalCluster;
import org.apache.storm.StormSubmitter;
import org.apache.storm.kafka.bolt.KafkaBolt;
import org.apache.storm.kafka.bolt.mapper.FieldNameBasedTupleToKafkaMapper;
import org.apache.storm.kafka.bolt.selector.DefaultTopicSelector;
import org.apache.storm.topology.TopologyBuilder;
import org.apache.storm.utils.Utils;

import java.util.Properties;
```

```
public class TopologyKafkaProducerSpout {
    //申请的ckafka实例ip:port
    private final static String BOOTSTRAP_SERVERS = "xx.xx.xx.xx:xxxx";
    //指定要将消息写入的topic
    private final static String TOPIC = "storm_test";
    public static void main(String[] args) throws Exception {
        //设置producer属性
        //函数参见：https://kafka.apache.org/0100/javadoc/index.html?org/apache/kafka
        //属性参见：http://kafka.apache.org/0102/documentation.html
        Properties properties = new Properties();
        properties.put("bootstrap.servers", BOOTSTRAP_SERVERS);
        properties.put("acks", "1");
        properties.put("key.serializer", "org.apache.kafka.common.serialization.Str
        properties.put("value.serializer", "org.apache.kafka.common.serialization.S

        //创建写入kafka的bolt，默认使用fields("key" "message")作为生产消息的key和message,
        KafkaBolt kafkaBolt = new KafkaBolt()
                .withProducerProperties(properties)
                .withTopicSelector(new DefaultTopicSelector(TOPIC))
                .withTupleToKafkaMapper(new FieldNameBasedTupleToKafkaMapper());
        TopologyBuilder builder = new TopologyBuilder();
        //一个顺序生成消息的spout类，输出field是sentence
        SerialSentenceSpout spout = new SerialSentenceSpout();
        AddMessageKeyBolt bolt = new AddMessageKeyBolt();
        builder.setSpout("kafka-spout", spout, 1);
        //为tuple加上生产到ckafka所需要的fields
        builder.setBolt("add-key", bolt, 1).shuffleGrouping("kafka-spout");
        //写入ckafka
        builder.setBolt("sendToKafka", kafkaBolt, 8).shuffleGrouping("add-key");

        Config config = new Config();
        if (args != null && args.length > 0) {
            //集群模式，用于打包jar，并放到storm运行
            config.setNumWorkers(1);
            StormSubmitter.submitTopologyWithProgressBar(args[0], config, builder.c
        } else {
            //本地模式
            LocalCluster cluster = new LocalCluster();
            cluster.submitTopology("test", config, builder.createTopology());
            Utils.sleep(10000);
            cluster.killTopology("test");
            cluster.shutdown();
        }

    }
}
```

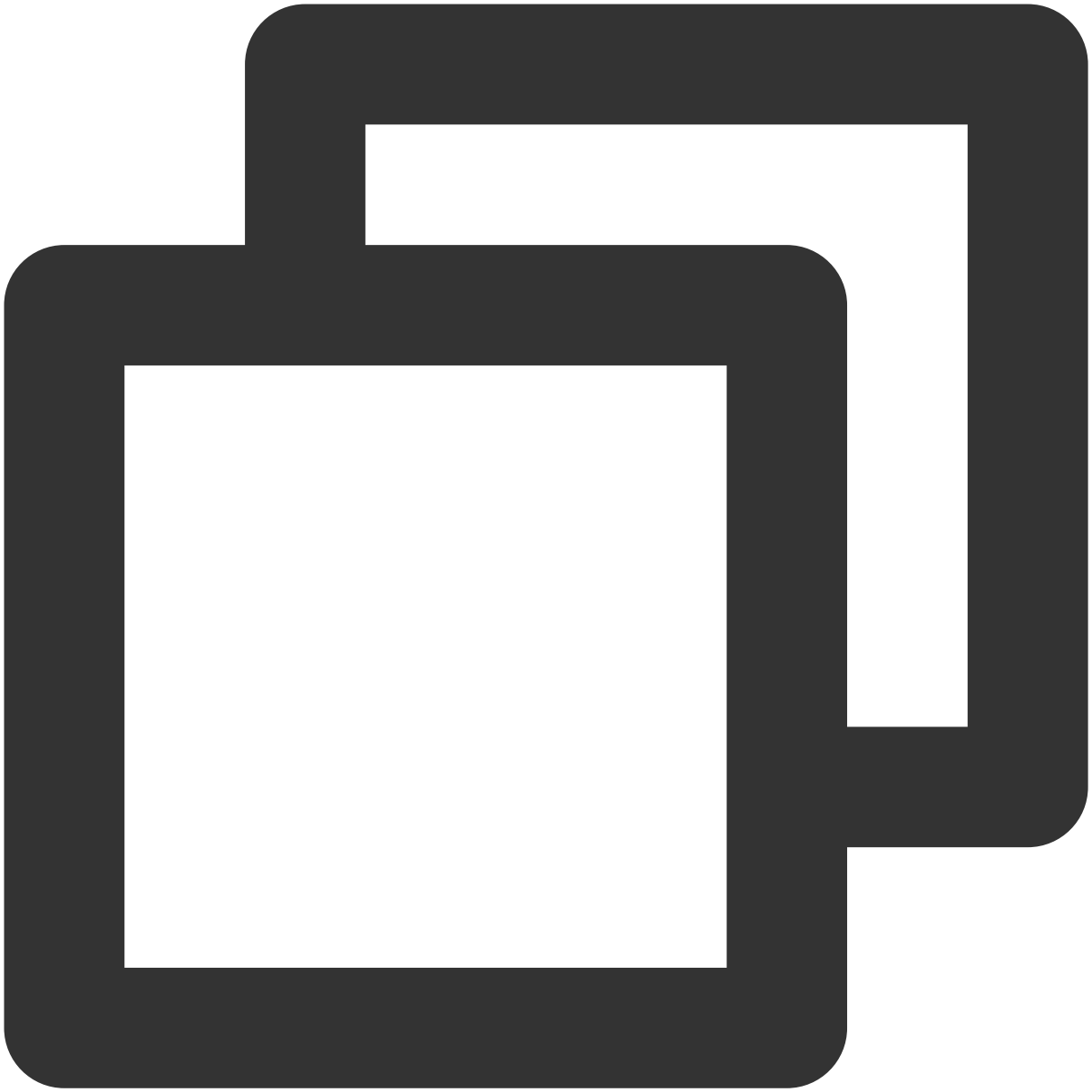
创建一个顺序生成消息的 spout 类：



```
import org.apache.storm.spout.SpoutOutputCollector;
import org.apache.storm.task.TopologyContext;
import org.apache.storm.topology.OutputFieldsDeclarer;
import org.apache.storm.topology.base.BaseRichSpout;
import org.apache.storm.tuple.Fields;
import org.apache.storm.tuple.Values;
import org.apache.storm.utils.Utils;
```

```
import java.util.Map;
import java.util.UUID;

public class SerialSentenceSpout extends BaseRichSpout {

    private SpoutOutputCollector spoutOutputCollector;

    @Override
    public void open(Map map, TopologyContext topologyContext, SpoutOutputCollector
        this.spoutOutputCollector = spoutOutputCollector;
    }

    @Override
    public void nextTuple() {
        Utils.sleep(1000);
        //生产一个UUID字符串发送给下一个组件
        spoutOutputCollector.emit(new Values(UUID.randomUUID().toString()));
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer outputFieldsDeclarer) {
        outputFieldsDeclarer.declare(new Fields("sentence"));
    }
}
```

为 `tuple` 加上 key、message 两个字段，当 key 为 null 时，生产的消息均匀分配到各个 partition，指定了 key 后将按照 key 值 hash 到特定 partition 上：

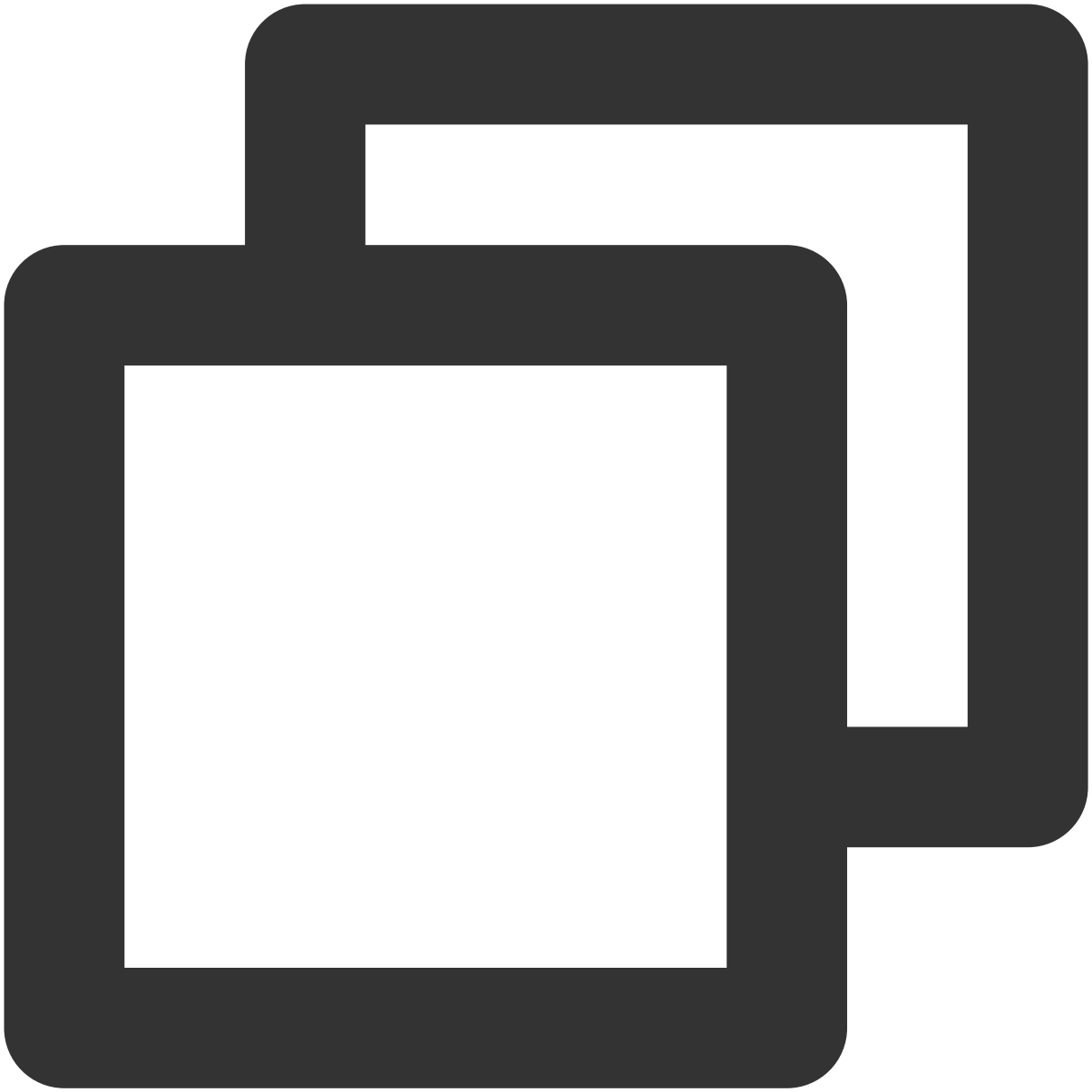


```
//AddMessageKeyBolt.java
import org.apache.storm.topology.BasicOutputCollector;
import org.apache.storm.topology.OutputFieldsDeclarer;
import org.apache.storm.topology.base.BaseBasicBolt;
import org.apache.storm.tuple.Fields;
import org.apache.storm.tuple.Tuple;
import org.apache.storm.tuple.Values;

public class AddMessageKeyBolt extends BaseBasicBolt {

    @Override
```

```
    public void execute(Tuple tuple, BasicOutputCollector basicOutputCollector) {
        //取出第一个filed值
        String messae = tuple.getString(0);
//        System.out.println(messae);
        //发送给下一个组件
        basicOutputCollector.emit(new Values(null, messae));
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer outputFieldsDeclarer) {
        //创建发送给下一个组件的schema
        outputFieldsDeclarer.declare(new Fields("key", "message"));
    }
}
```

使用 trident

使用 trident 类生成 topology：

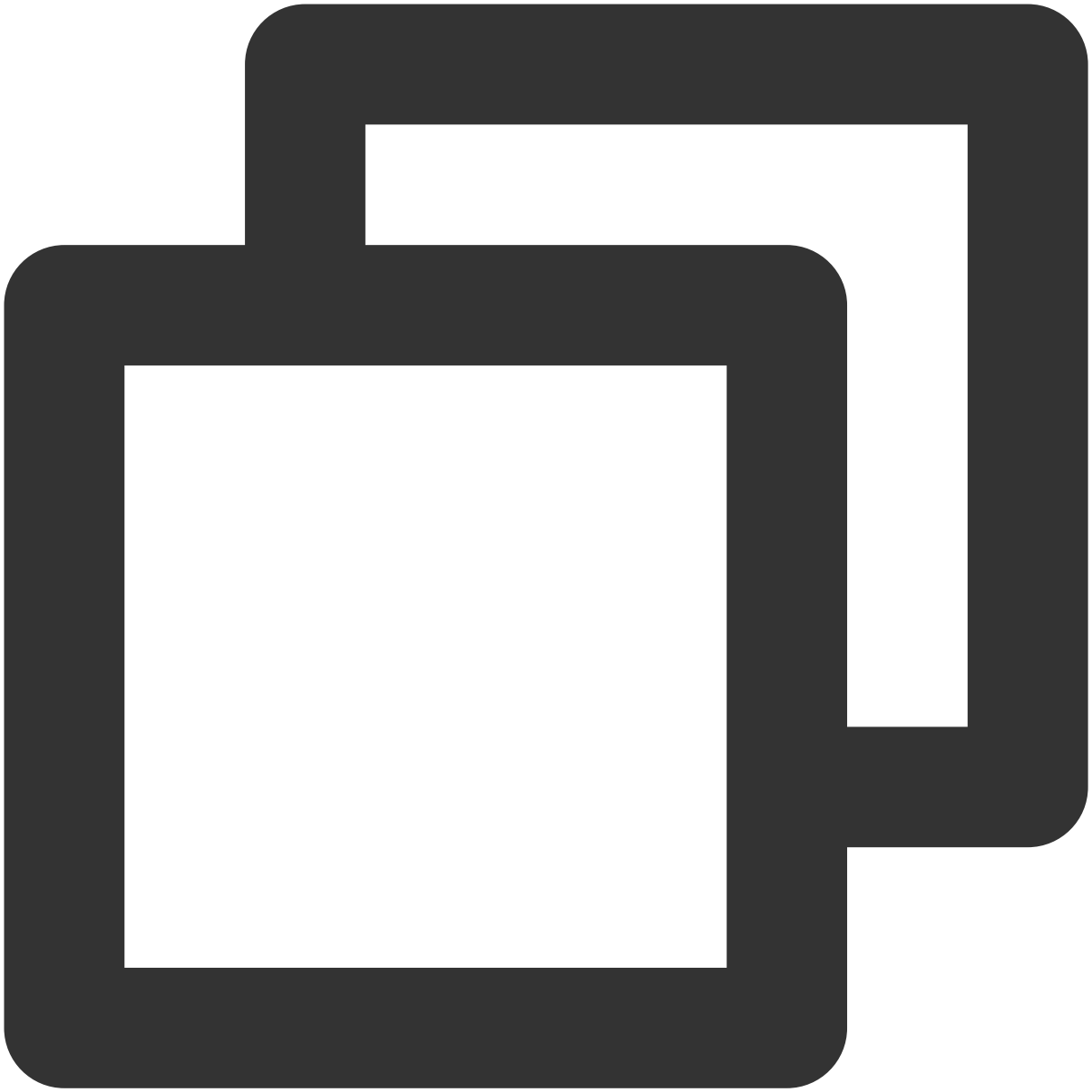


```
//TopologyKafkaProducerTrident.java
import org.apache.storm.Config;
import org.apache.storm.LocalCluster;
import org.apache.storm.StormSubmitter;
import org.apache.storm.kafka.trident.TridentKafkaStateFactory;
import org.apache.storm.kafka.trident.TridentKafkaStateUpdater;
import org.apache.storm.kafka.trident.mapper.FieldNameBasedTupleToKafkaMapper;
import org.apache.storm.kafka.trident.selector.DefaultTopicSelector;
import org.apache.storm.trident.TridentTopology;
import org.apache.storm.trident.operation.BaseFunction;
import org.apache.storm.trident.operation.TridentCollector;
```

```
import org.apache.storm.trident.tuple.TridentTuple;
import org.apache.storm.tuple.Fields;
import org.apache.storm.tuple.Values;
import org.apache.storm.utils.Utils;

import java.util.Properties;

public class TopologyKafkaProducerTrident {
    //申请的ckafka实例ip:port
    private final static String BOOTSTRAP_SERVERS = "xx.xx.xx.xx:xxxx";
    //指定要将消息写入的topic
    private final static String TOPIC = "storm_test";
    public static void main(String[] args) throws Exception {
        //设置producer属性
        //函数参见：https://kafka.apache.org/0100/javadoc/index.html?org/apache/kafka
        //属性参见：http://kafka.apache.org/0102/documentation.html
        Properties properties = new Properties();
        properties.put("bootstrap.servers", BOOTSTRAP_SERVERS);
        properties.put("acks", "1");
        properties.put("key.serializer", "org.apache.kafka.common.serialization.Str
        properties.put("value.serializer", "org.apache.kafka.common.serialization.S
        //设置Trident
        TridentKafkaStateFactory stateFactory = new TridentKafkaStateFactory()
                .withProducerProperties(properties)
                .withKafkaTopicSelector(new DefaultTopicSelector(TOPIC))
                //设置使用fields("key", "value")作为消息写入  不像FieldNameBasedTupleTo
                .withTridentTupleToKafkaMapper(new FieldNameBasedTupleToKafkaMapper
        TridentTopology builder = new TridentTopology();
        //一个批量产生句子的spout,输出field为sentence
        builder.newStream("kafka-spout", new TridentSerialSentenceSpout(5))
                .each(new Fields("sentence"), new AddMessageKey(), new Fields("key"
                .partitionPersist(stateFactory, new Fields("key", "value"), new Tri

        Config config = new Config();
        if (args != null && args.length > 0) {
            //集群模式，用于打包jar，并放到storm运行
            config.setNumWorkers(1);
            StormSubmitter.submitTopologyWithProgressBar(args[0], config, builder.b
        } else {
            //本地模式
            LocalCluster cluster = new LocalCluster();
            cluster.submitTopology("test", config, builder.build());
            Utils.sleep(10000);
            cluster.killTopology("test");
            cluster.shutdown();
        }
```

```
    }

    private static class AddMessageKey extends BaseFunction {

        @Override
        public void execute(TridentTuple tridentTuple, TridentCollector tridentColl
            //取出第一个filed值
            String messae = tridentTuple.getString(0);
            //System.out.println(messae);
            //发送给下一个组件
            //tridentCollector.emit(new Values(Integer.toString(messae.hashCode()),
            tridentCollector.emit(new Values(null, messae));
        }
    }
}
```

创建一个批量生成消息的 spout 类：

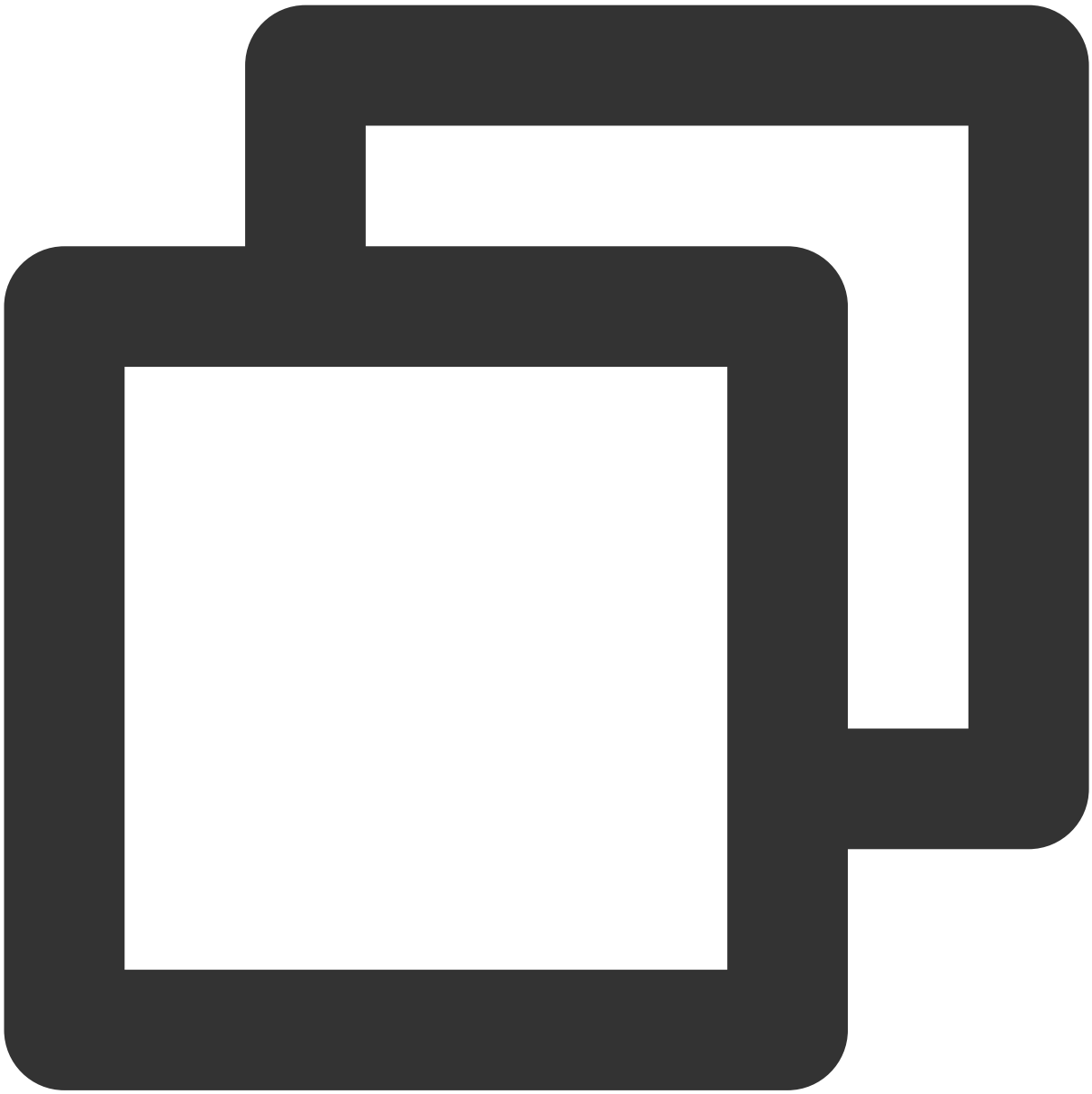


```
//TridentSerialSentenceSpout.java
import org.apache.storm.Config;
import org.apache.storm.task.TopologyContext;
import org.apache.storm.trident.operation.TridentCollector;
import org.apache.storm.trident.spout.IBatchSpout;
import org.apache.storm.tuple.Fields;
import org.apache.storm.tuple.Values;
import org.apache.storm.utils.Utils;

import java.util.Map;
import java.util.UUID;
```

```
public class TridentSerialSentenceSpout implements IBatchSpout {

    private final int batchCount;

    public TridentSerialSentenceSpout(int batchCount) {
        this.batchCount = batchCount;
    }

    @Override
    public void open(Map map, TopologyContext topologyContext) {

    }

    @Override
    public void emitBatch(long l, TridentCollector tridentCollector) {
        Utils.sleep(1000);
        for(int i = 0; i < batchCount; i++){
            tridentCollector.emit(new Values(UUID.randomUUID().toString()));
        }
    }

    @Override
    public void ack(long l) {

    }

    @Override
    public void close() {

    }

    @Override
    public Map<String, Object> getComponentConfiguration() {
        Config conf = new Config();
        conf.setMaxTaskParallelism(1);
        return conf;
    }

    @Override
    public Fields getOutputFields() {
        return new Fields("sentence");
    }
}
```

步骤5：消费消息

使用 spout/bolt

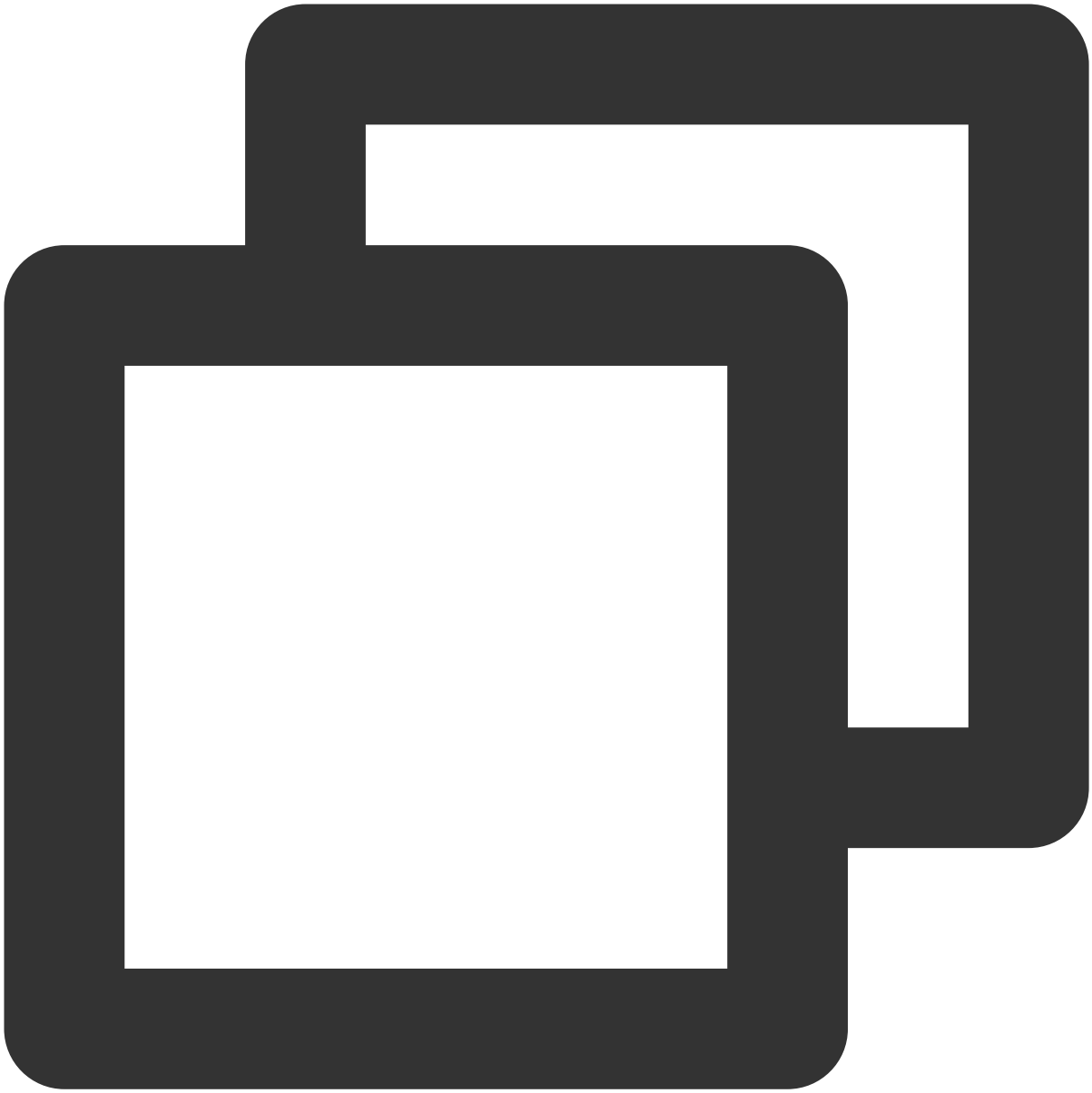


```
//TopologyKafkaConsumerSpout.java
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.storm.Config;
import org.apache.storm.LocalCluster;
import org.apache.storm.StormSubmitter;
import org.apache.storm.kafka.spout.*;
import org.apache.storm.task.OutputCollector;
```

```
import org.apache.storm.task.TopologyContext;
import org.apache.storm.topology.OutputFieldsDeclarer;
import org.apache.storm.topology.TopologyBuilder;
import org.apache.storm.topology.base.BaseRichBolt;
import org.apache.storm.tuple.Fields;
import org.apache.storm.tuple.Tuple;
import org.apache.storm.tuple.Values;
import org.apache.storm.utils.Utils;

import java.util.HashMap;
import java.util.Map;

import static org.apache.storm.kafka.spout.FirstPollOffsetStrategy.LATEST;

public class TopologyKafkaConsumerSpout {
    //申请的ckafka实例ip:port
    private final static String BOOTSTRAP_SERVERS = "xx.xx.xx.xx:xxxx";
    //指定要将消息写入的topic
    private final static String TOPIC = "storm_test";

    public static void main(String[] args) throws Exception {
        //设置重试策略
        KafkaSpoutRetryService kafkaSpoutRetryService = new KafkaSpoutRetryExponent
                KafkaSpoutRetryExponentialBackoff.TimeInterval.microSeconds(500),
                KafkaSpoutRetryExponentialBackoff.TimeInterval.milliSeconds(2),
                Integer.MAX_VALUE,
                KafkaSpoutRetryExponentialBackoff.TimeInterval.seconds(10)
        );
        ByTopicRecordTranslator<String, String> trans = new ByTopicRecordTranslator
                (r) -> new Values(r.topic(), r.partition(), r.offset(), r.key(), r.
                new Fields("topic", "partition", "offset", "key", "value"));
        //设置consumer参数
        //函数参见http://storm.apache.org/releases/1.1.0/javadocs/org/apache/storm/ka
        //参数参见http://kafka.apache.org/0102/documentation.html
        KafkaSpoutConfig spoutConfig = KafkaSpoutConfig.builder(BOOTSTRAP_SERVERS,
                .setProp(new HashMap<String, Object>(){{
                    put(ConsumerConfig.GROUP_ID_CONFIG, "test-group1"); //设置group
                    put(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG, "50000"); //设置se
                    put(ConsumerConfig.REQUEST_TIMEOUT_MS_CONFIG, "60000"); //设置请
                }})
                .setOffsetCommitPeriodMs(10_000) //设置自动确认时间
                .setFirstPollOffsetStrategy(LATEST) //设置拉取最新消息
                .setRetry(kafkaSpoutRetryService)
                .setRecordTranslator(trans)
                .build();

        TopologyBuilder builder = new TopologyBuilder();
```

```
        builder.setSpout("kafka-spout", new KafkaSpout(spoutConfig), 1);
        builder.setBolt("bolt", new BaseRichBolt(){
            private OutputCollector outputCollector;
            @Override
            public void declareOutputFields(OutputFieldsDeclarer outputFieldsDeclar

            }

            @Override
            public void prepare(Map map, TopologyContext topologyContext, OutputCol
                this.outputCollector = outputCollector;
            }

            @Override
            public void execute(Tuple tuple) {
                System.out.println(tuple.getStringByField("value"));
                outputCollector.ack(tuple);
            }
        }, 1).shuffleGrouping("kafka-spout");

        Config config = new Config();
        config.setMaxSpoutPending(20);
        if (args != null && args.length > 0) {
            config.setNumWorkers(3);
            StormSubmitter.submitTopologyWithProgressBar(args[0], config, builder.c
        }
        else {
            LocalCluster cluster = new LocalCluster();
            cluster.submitTopology("test", config, builder.createTopology());
            Utils.sleep(20000);
            cluster.killTopology("test");
            cluster.shutdown();
        }
    }
}
```

使用 trident



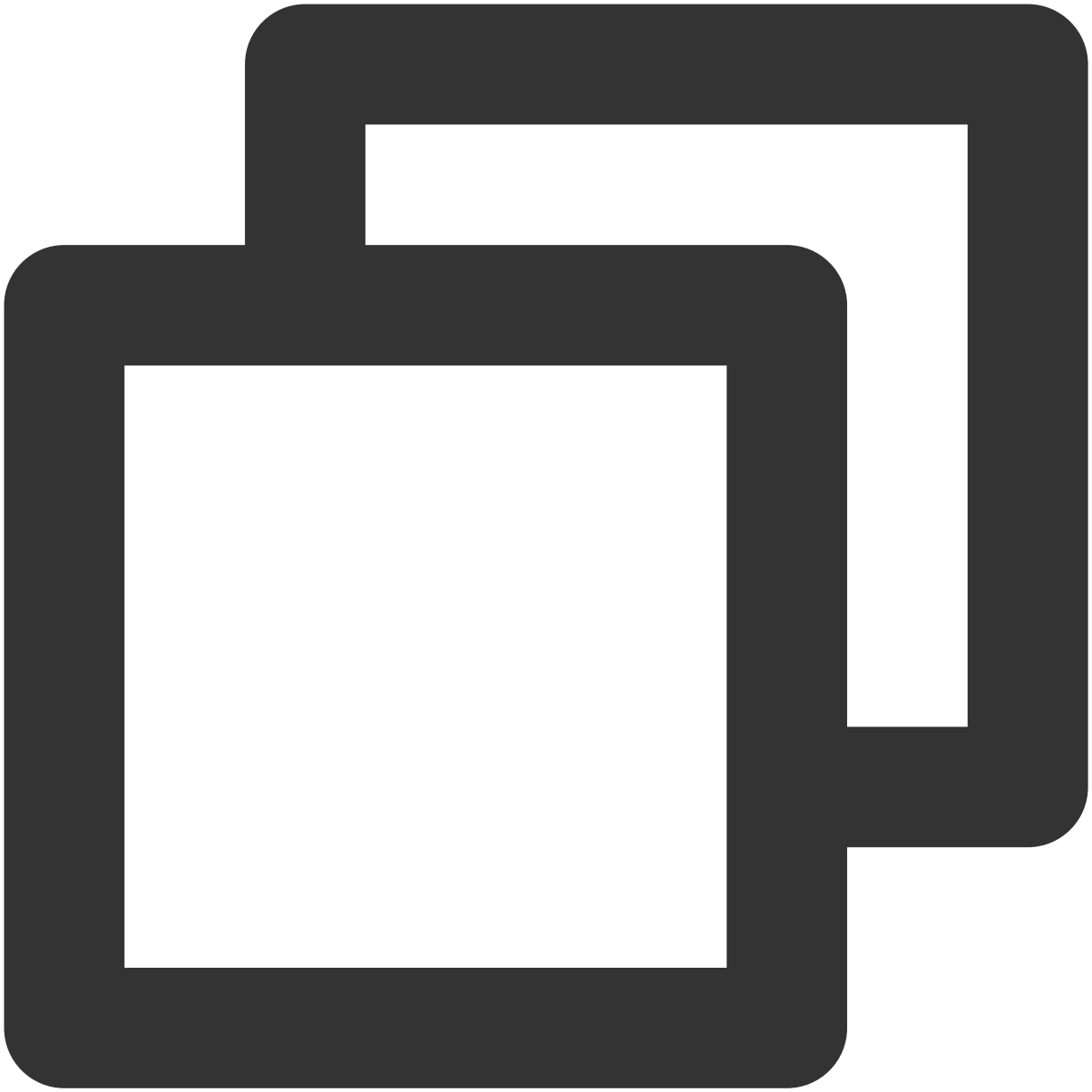
```
//TopologyKafkaConsumerTrident.java
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.storm.Config;
import org.apache.storm.LocalCluster;
import org.apache.storm.StormSubmitter;
import org.apache.storm.generated.StormTopology;
import org.apache.storm.kafka.spout.ByTopicRecordTranslator;
import org.apache.storm.kafka.spout.trident.KafkaTridentSpoutConfig;
import org.apache.storm.kafka.spout.trident.KafkaTridentSpoutOpaque;
import org.apache.storm.trident.Stream;
import org.apache.storm.trident.TridentTopology;
```

```
import org.apache.storm.trident.operation.BaseFunction;
import org.apache.storm.trident.operation.TridentCollector;
import org.apache.storm.trident.tuple.TridentTuple;
import org.apache.storm.tuple.Fields;
import org.apache.storm.tuple.Values;
import org.apache.storm.utils.Utils;

import java.util.HashMap;

import static org.apache.storm.kafka.spout.FirstPollOffsetStrategy.LATEST;


public class TopologyKafkaConsumerTrident {
    //申请的ckafka实例ip:port
    private final static String BOOTSTRAP_SERVERS = "xx.xx.xx.xx:xxxx";
    //指定要将消息写入的topic
    private final static String TOPIC = "storm_test";

    public static void main(String[] args) throws Exception {
        ByTopicRecordTranslator<String, String> trans = new ByTopicRecordTranslator
                (r) -> new Values(r.topic(), r.partition(), r.offset(), r.key(), r.
                new Fields("topic", "partition", "offset", "key", "value"));
        //设置consumer参数
        //函数参见http://storm.apache.org/releases/1.1.0/javadocs/org/apache/storm/ka
        //参数参见http://kafka.apache.org/0102/documentation.html
        KafkaTridentSpoutConfig spoutConfig = KafkaTridentSpoutConfig.builder(BOOTS
                .setProp(new HashMap<String, Object>(){{
                    put(ConsumerConfig.GROUP_ID_CONFIG, "test-group1"); //设置group
                    put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "true"); //设置自动
                    put(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG, "50000"); //设置se
                    put(ConsumerConfig.REQUEST_TIMEOUT_MS_CONFIG, "60000"); //设置请
                }})
                .setFirstPollOffsetStrategy(LATEST) //设置拉取最新消息
                .setRecordTranslator(trans)
                .build();

        TridentTopology builder = new TridentTopology();
//      Stream spoutStream = builder.newStream("spout", new KafkaTridentSpoutTransa
        Stream spoutStream = builder.newStream("spout", new KafkaTridentSpoutOpaque
        spoutStream.each(spoutStream.getOutputFields(), new BaseFunction(){
            @Override
            public void execute(TridentTuple tridentTuple, TridentCollector trident
                System.out.println(tridentTuple.getStringByField("value"));
                tridentCollector.emit(new Values(tridentTuple.getStringByField("val
            }
        }, new Fields("message"));
```

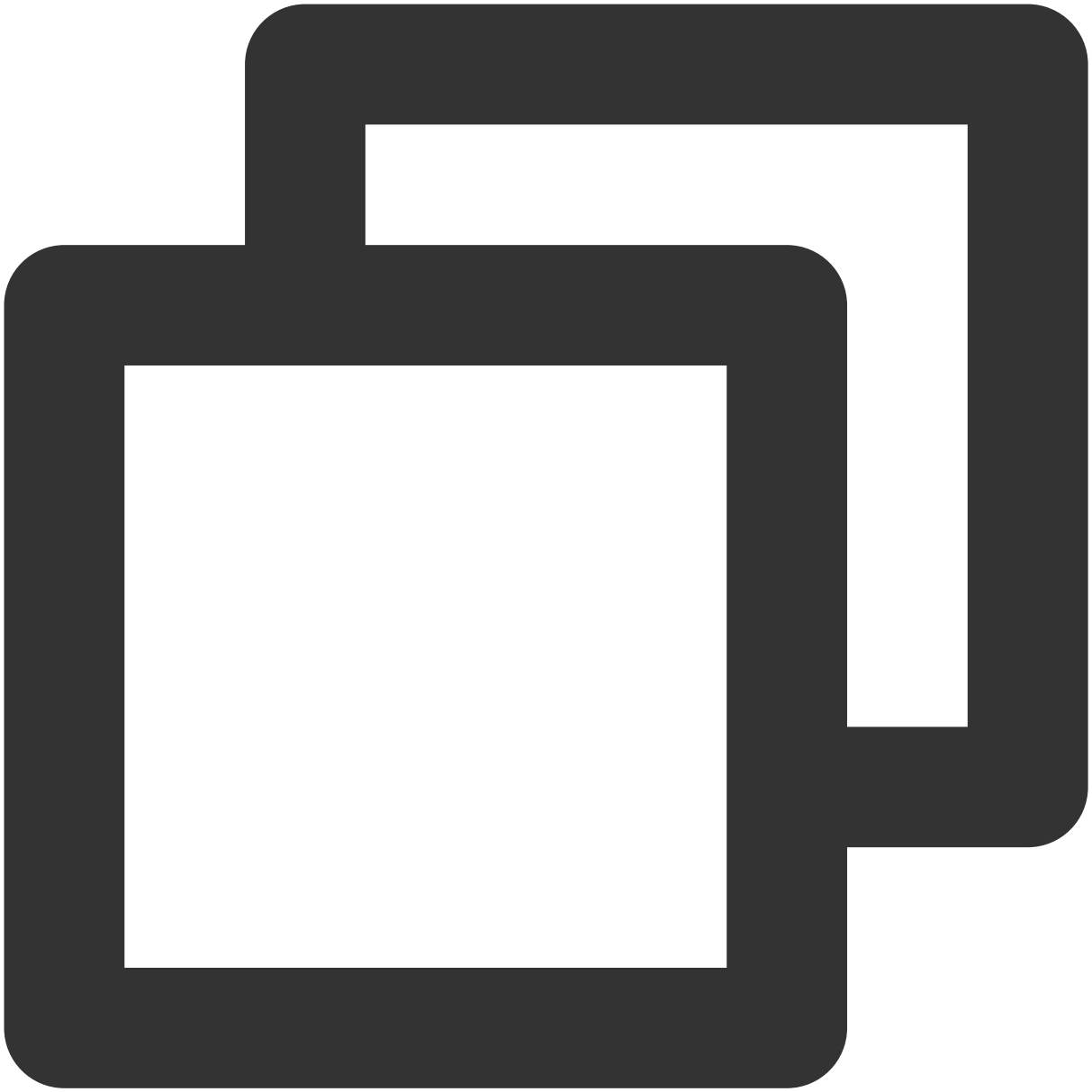
```
        Config conf = new Config();
        conf.setMaxSpoutPending(20);conf.setNumWorkers(1);
        if (args != null && args.length > 0) {
            conf.setNumWorkers(3);
            StormSubmitter.submitTopologyWithProgressBar(args[0], conf, builder.bui
        }
        else {
            StormTopology stormTopology = builder.build();
            LocalCluster cluster = new LocalCluster();
            cluster.submitTopology("test", conf, stormTopology);
            Utils.sleep(10000);
            cluster.killTopology("test");
            cluster.shutdown();stormTopology.clear();
        }
    }
}
```

步骤6：提交 Storm

使用 mvn package 编译后，可以提交到本地集群进行 debug 测试，也可以提交到正式集群进行运行。



```
storm jar your_jar_name.jar topology_name
```



```
storm jar your_jar_name.jar topology_name tast_name
```

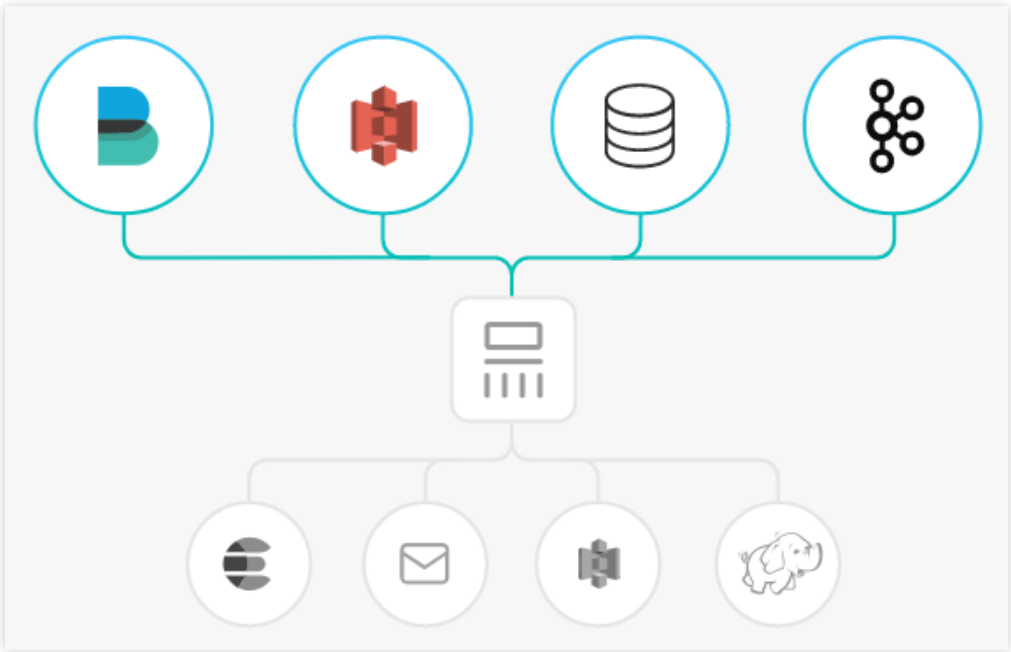
Logstash 接入 CKafka

最近更新时间：2024-05-31 12:08:45

Logstash 是一个开源的日志处理工具，可以从多个源头收集数据、过滤收集的数据并对数据进行存储作为其他用途。
Logstash 灵活性强，拥有强大的语法分析功能，插件丰富，支持多种输入和输出源。Logstash 作为水平可伸缩的数据管道，与 Elasticsearch 和 Kibana 配合，在日志收集检索方面功能强大。

Logstash 工作原理

Logstash 数据处理可以分为三个阶段：inputs → filters → outputs。
1. inputs：产生数据来源，例如文件、syslog、redis 和 beats 此类来源。
2. filters：修改过滤数据， 在 Logstash 数据管道中属于中间环节，可以根据条件去对事件进行更改。一些常见的过滤器包括：grok、mutate、drop 和 clone 等。
3. outputs：将数据传输到其他地方，一个事件可以传输到多个 outputs，当传输完成后这个事件就结束。Elasticsearch 就是最常见的 outputs。
同时 Logstash 支持编码解码，可以在 inputs 和 outputs 端指定格式。

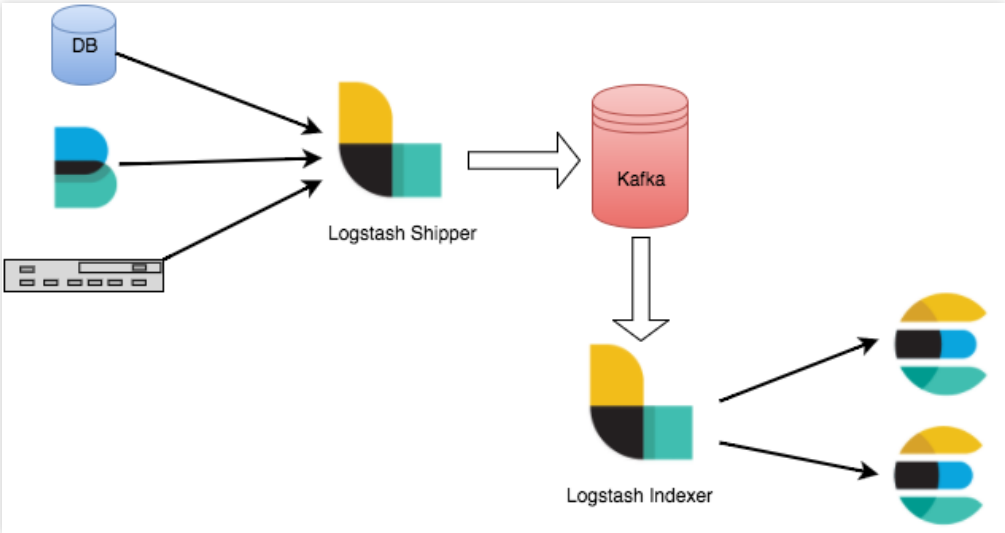


Logstash 接入 Kafka 的优势

可以异步处理数据：防止突发流量。
解耦：当 Elasticsearch 异常的时候不会影响上游工作。

注意：

Logstash 过滤消耗资源，如果部署在生产 server 上会影响其性能。



操作步骤

准备工作

下载并安装 Logstash，参见 Download Logstash。

下载并安装 JDK 8，参见 Download JDK 8。

已 创建 CKafka 实例。

步骤1：获取 CKafka 实例接入地址

1. 登录 CKafka 控制台。
2. 在左侧导航栏选择**实例列表**，单击实例的“ID”，进入实例基本信息页面。
3. 在实例的基本信息页面的**接入方式**模块，可获取实例的接入地址。

步骤2：创建 Topic

1. 在实例基本信息页面，选择顶部**Topic管理**页签。
2. 在 Topic 管理页面，单击**新建**，创建一个名为 logstash_test 的 Topic。

ID/名称	监控	分区数(个)	副本数(个)	白名单	备注	创建时间
topic-mi1h60v0 logstash_test		1	2	未开启		2021-07-13 19:53:59

步骤3：接入 CKafka

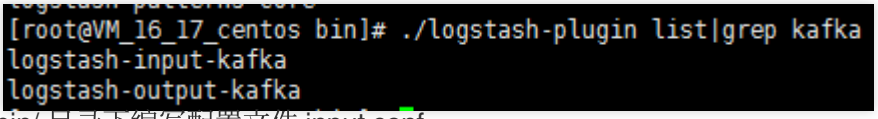
说明：

您可以单击以下页签，查看 CKafka 作为 inputs 或者 outputs 接入的具体步骤。

作为 inputs 接入

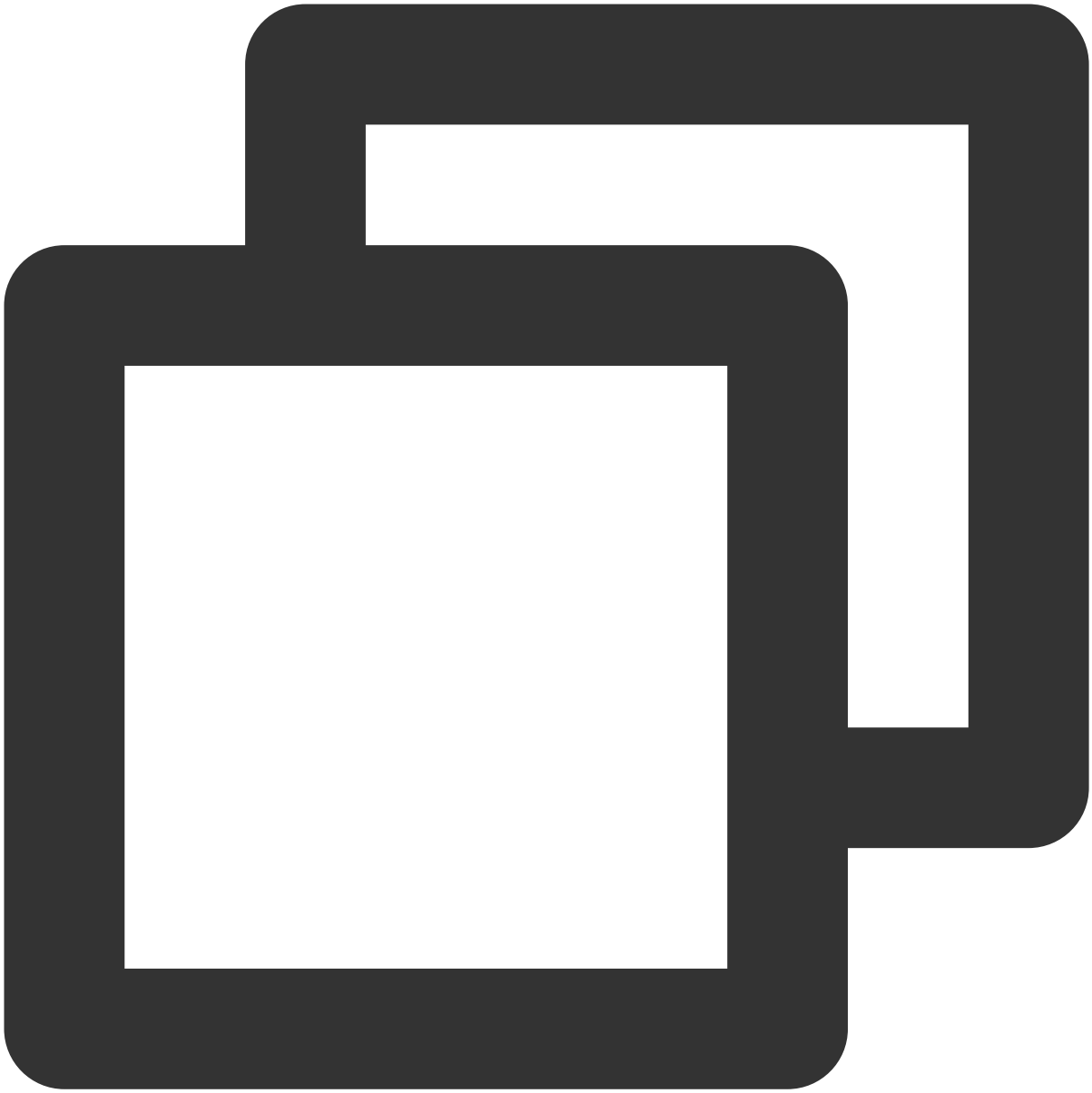
作为 outputs 接入

1. 执行 `bin/logstash-plugin list` ，查看已经支持的插件是否含有 logstash-input-kafka。



2. 在 .bin/ 目录下编写配置文件 input.conf。

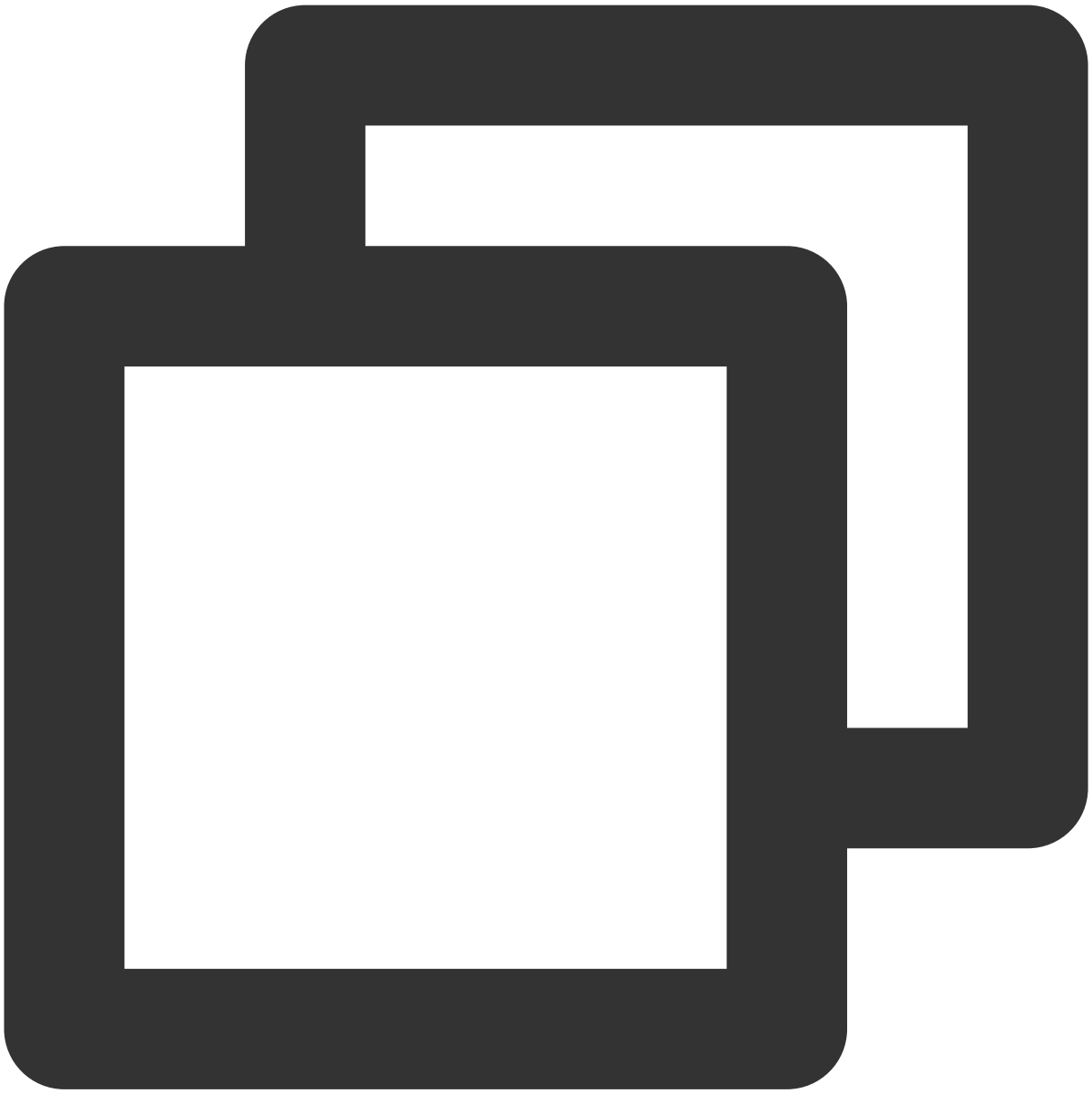
此处将标准输出作为数据终点，将 Kafka 作为数据来源。



```
input {
    kafka {
        bootstrap_servers => "xx.xx.xx.xx:xxxx" // ckafka 实例接入地址
        group_id => "logstash_group"  // ckafka groupid 名称
        topics => ["logstash_test"] // ckafka topic 名称
        consumer_threads => 3 // 消费线程数，一般与 ckafka 分区数一致
        auto_offset_reset => "earliest"
    }
}
output {
    stdout{codec=>rubydebug}
```

```
}
```

3. 执行以下命令启动 Logstash，进行消息消费。



```
./logstash -f input.conf
```

返回结果如下：

```
[root@VM_16_17_centos bin]# ./logstash -f input.conf
ERROR StatusLogger No log4j2 configuration file found. Using default configuration: logging only errors to
Sending Logstash's logs to /data/ryan/logstash-5.5.2/logs which is now configured via log4j2.properties
[2017-09-06T18:07:41,926][INFO ][logstash.pipeline        ] Starting pipeline {"id"=>"main", "pipeline.work
[2017-09-06T18:07:41,943][INFO ][logstash.pipeline        ] Pipeline main started
[2017-09-06T18:07:41,999][INFO ][logstash.agent           ] Successfully started Logstash API endpoint {:p
{
    "@timestamp" => 2017-09-06T10:07:42.256Z,
      "@version" => "1",
       "message" => "2017-09-06T09:24:23.039Z localhost ckafka"
}
{
    "@timestamp" => 2017-09-06T10:07:42.256Z,
      "@version" => "1",
       "message" => "2017-09-06T09:23:28.343Z localhost logstash"
}
{
    "@timestamp" => 2017-09-06T10:07:42.256Z,
      "@version" => "1",
       "message" => "2017-09-06T09:23:15.848Z localhost test"
}
```

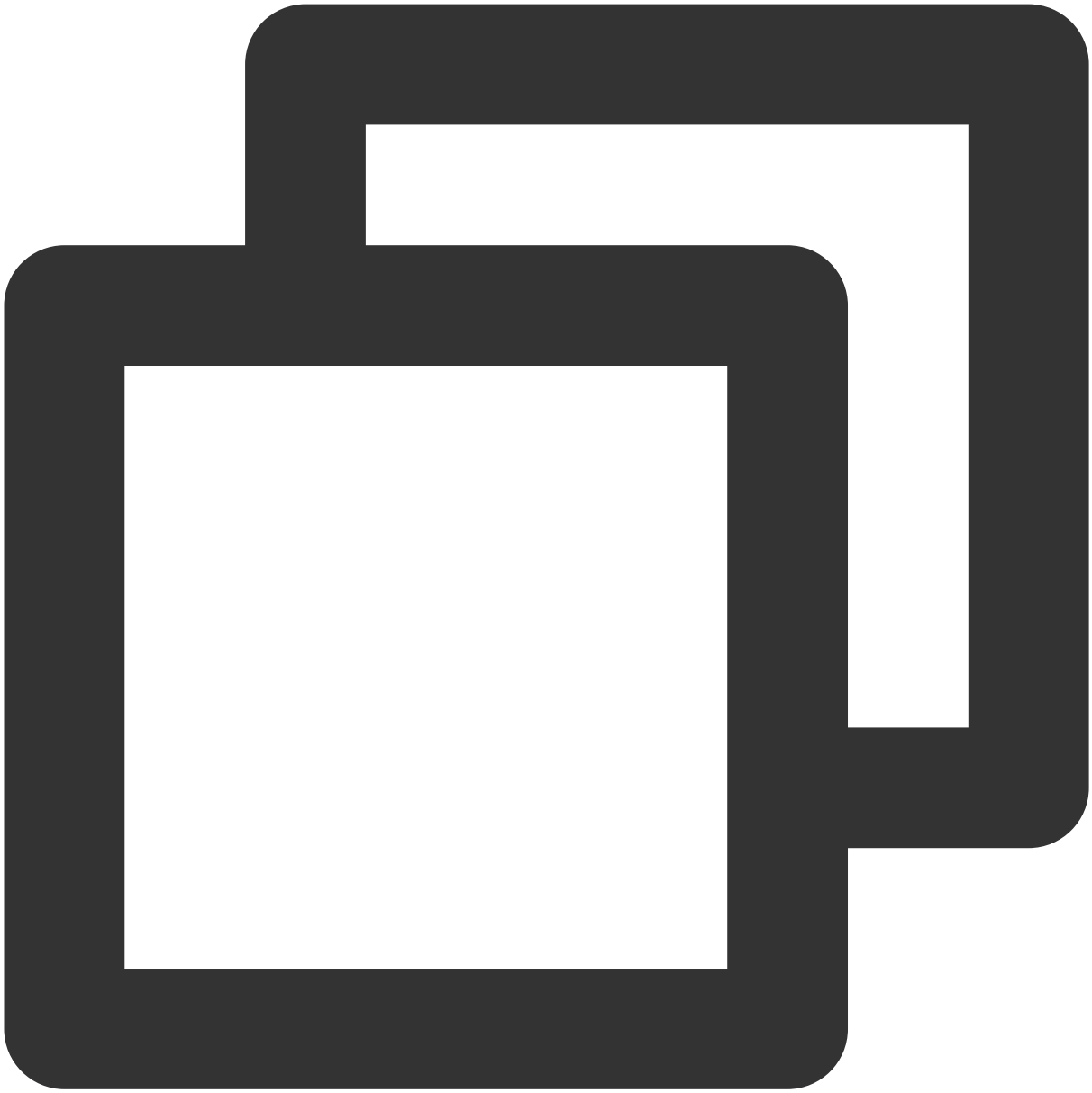
可以看到刚才 Topic 中的数据被消费出来。

1. 执行 `bin/logstash-plugin list` ，查看已经支持的插件是否含有 logstash-output-kafka。

```
[root@VM_16_17_centos bin]# ./logstash-plugin list|grep kafka
logstash-input-kafka
logstash-output-kafka
```

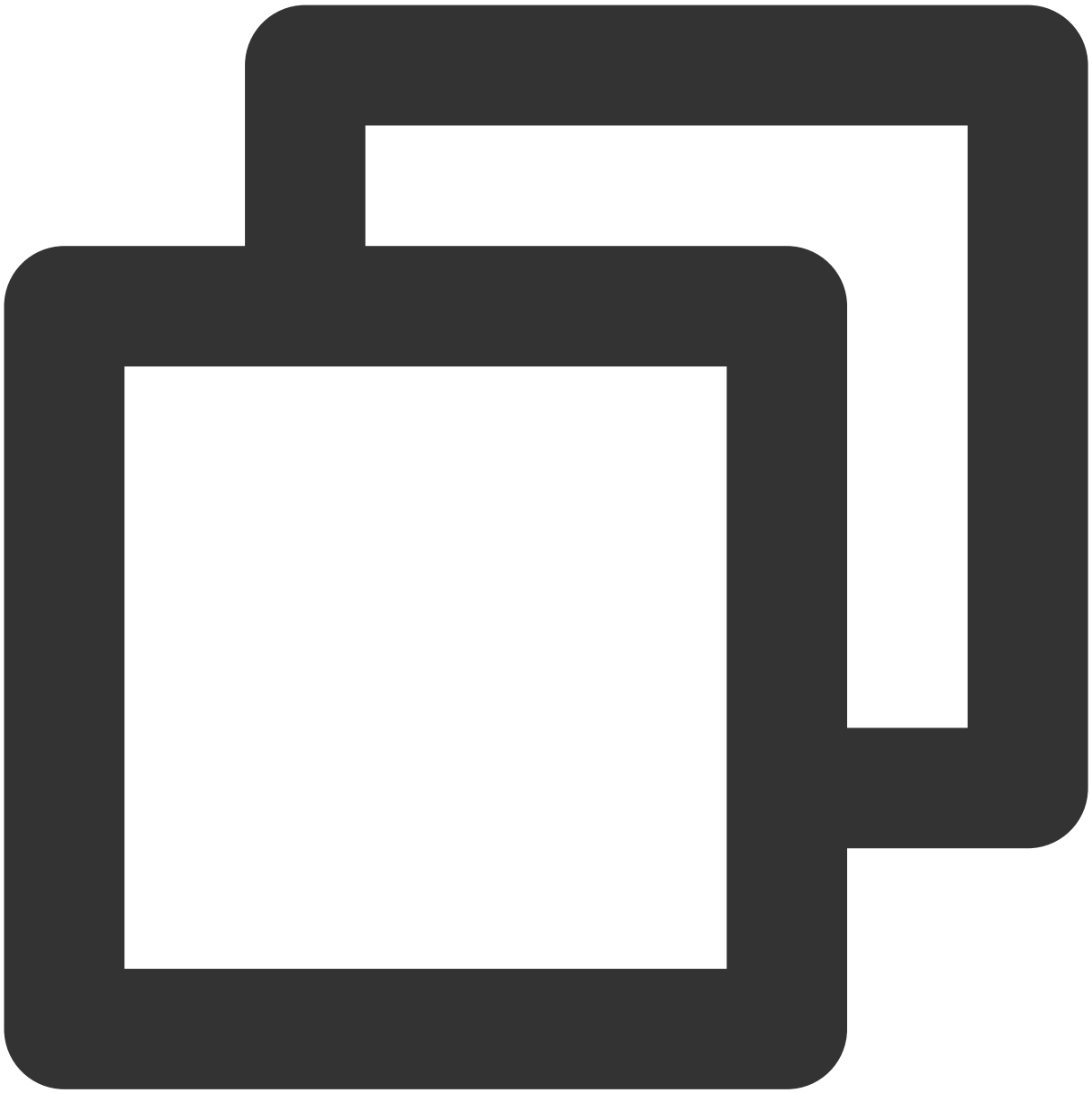
2. 在.bin/目录下编写配置文件 output.conf。

此处将标准输入作为数据来源，将 Kafka 作为数据目的地。

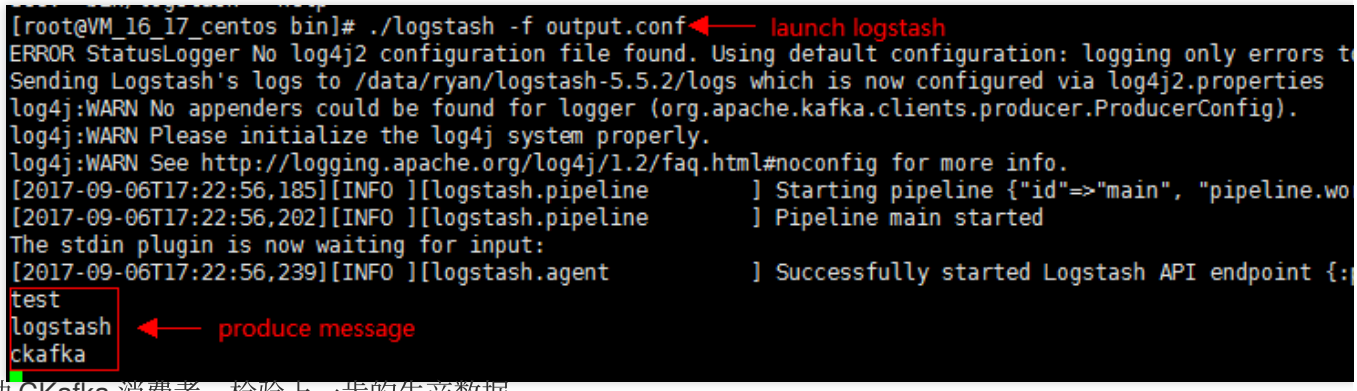


```
  input {
 stdin{}
}
output {
kafka {
   bootstrap_servers => "xx.xx.xx.xx:xxxx"  // ckafka 实例接入地址
   topic_id => "logstash_test" // ckafka topic 名称
  }
}
```

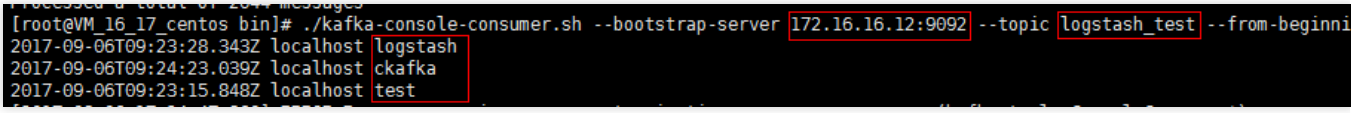
3. 执行如下命令启动 Logstash，向创建的 Topic 发送消息。



```
./logstash -f output.conf
```



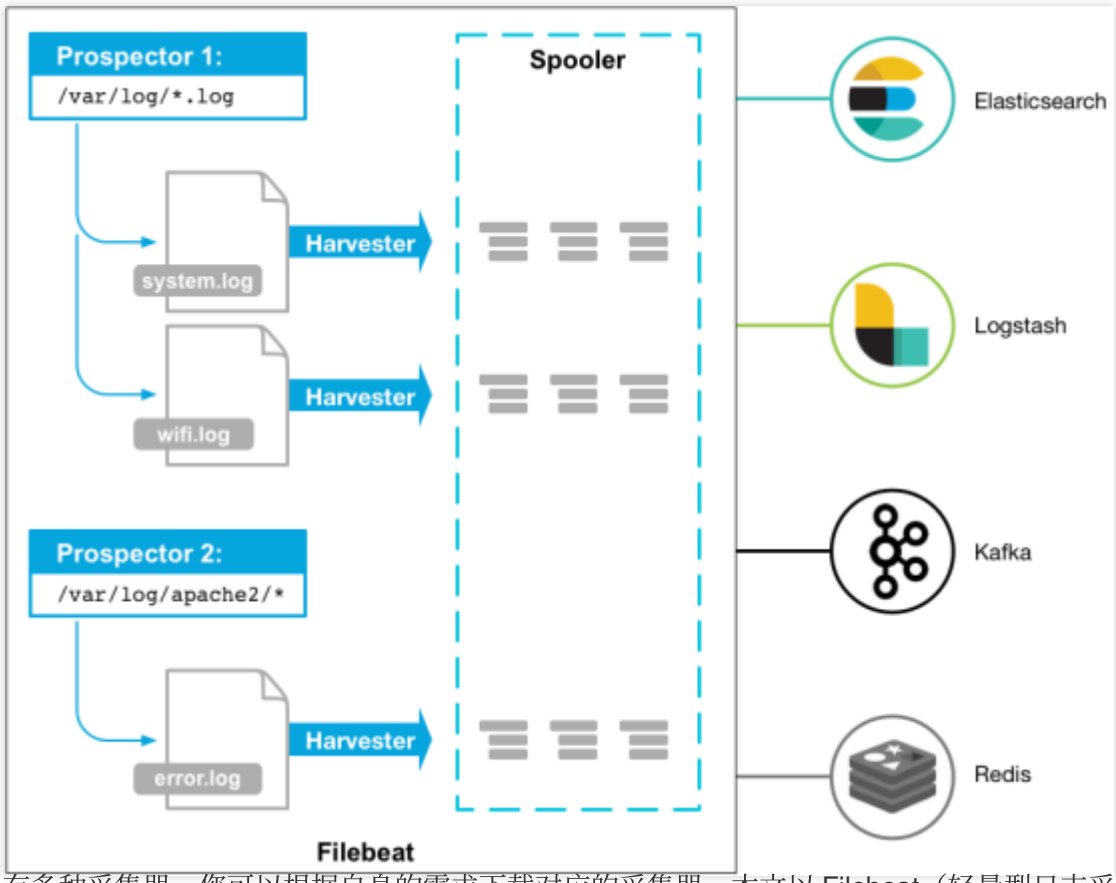
4. 启动 CKafka 消费者，检验上一步的生产数据。



Filebeat 接入 CKafka

最近更新时间：2024-05-31 12:08:45

Beats 平台 集合了多种单一用途数据采集器。这些采集器安装后可用作轻量型代理，从成百上千或成千上万台机器向目标发送采集数据。



Beats 有多种采集器，您可以根据自身的需求下载对应的采集器。本文以 Filebeat（轻量型日志采集器）为例，向您介绍 Filebeat 接入 CKafka 的操作方法，及接入后常见问题的解决方法。

前提条件

下载并安装 Filebeat（参见 Download Filebeat）
下载并安装JDK 8（参见 Download JDK 8）
已 创建 CKafka 实例

操作步骤

步骤1：获取 CKafka 实例接入地址

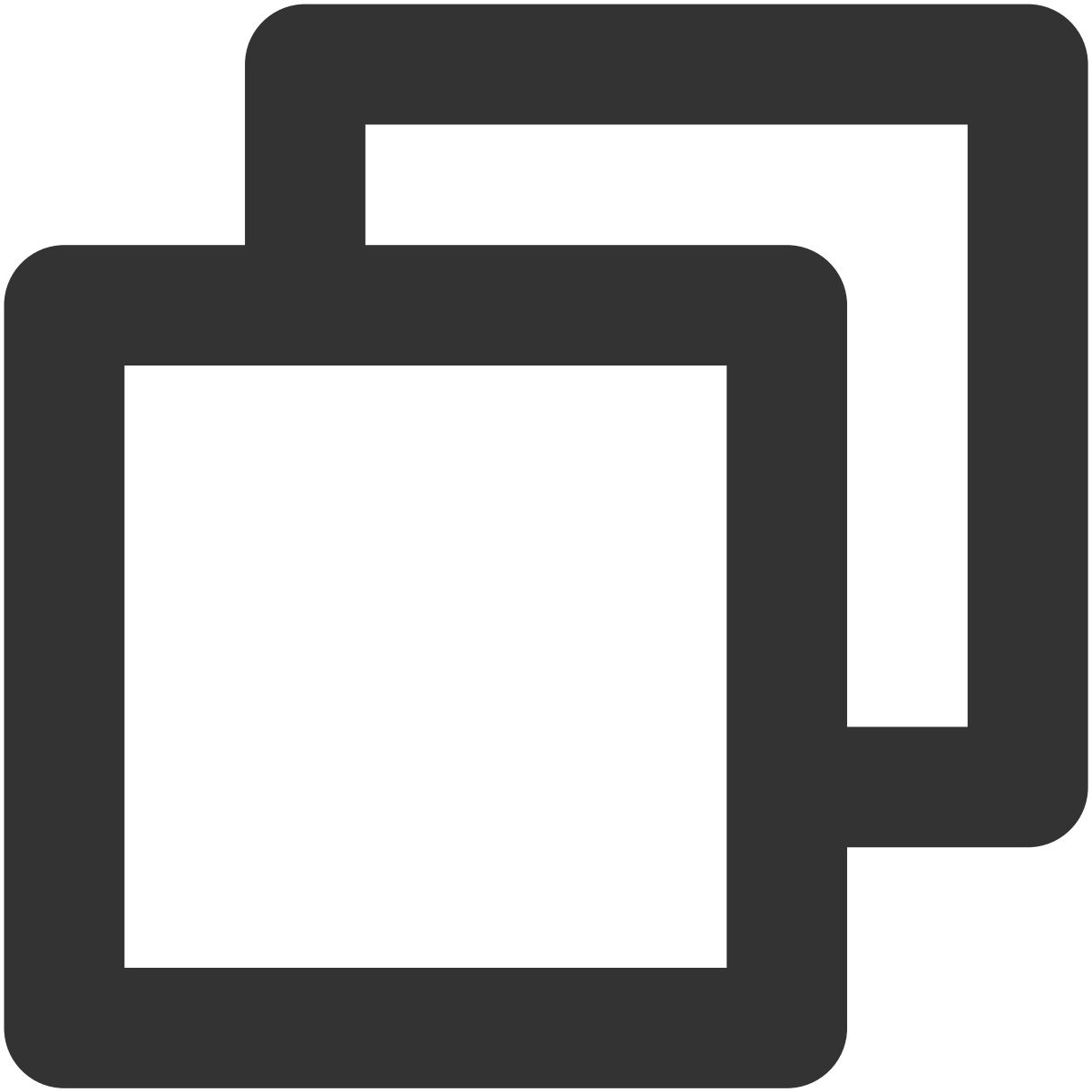
1. 登录 CKafka 控制台。
2. 在左侧导航栏选择**实例列表**，单击实例的“ID”，进入实例基本信息页面。
3. 在实例的基本信息页面的**接入方式**模块，可获取实例的接入地址。

步骤2：创建 Topic

1. 在实例基本信息页面，选择顶部**Topic管理**页签。
2. 在 Topic 管理页面，单击**新建**，创建一个名为 test 的 Topic。

步骤3：准备配置文件

进入 Filebeat 的安装目录，创建配置监控文件 filebeat.yml。

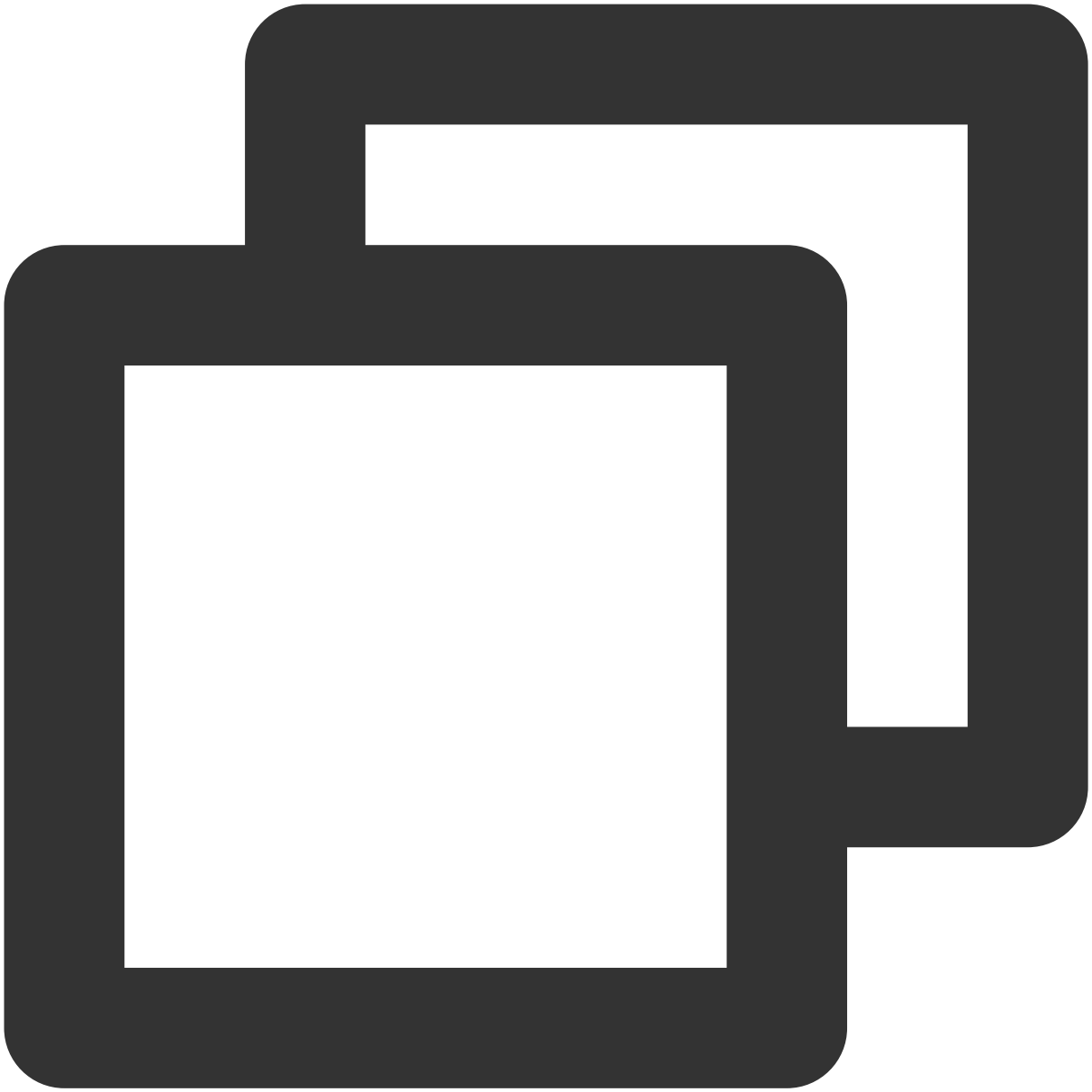


```
#======= Filebeat7.x之后的版本，将 filebeat.prospectors 修改为 filebeat.inputs 即可 ===
filebeat.prospectors:

- input_type: log

# 此处为监听文件路径
  paths:
    - /var/log/messages

#=======  Outputs =========
```

```
#------------------ kafka -------------------------------------
output.kafka:
  version:0.10.2 // 根据不同 CKafka 实例开源版本配置
  # 设置为CKafka实例的接入地址
  hosts: ["xx.xx.xx.xx:xxxx"]
  # 设置目标topic的名称
  topic: 'test'
  partition.round_robin:
    reachable_only: false

  required_acks: 1
  compression: none
  max_message_bytes: 1000000

  # SASL 需要配置下列信息，如果不需要则下面两个选项可不配置
  username: "yourinstance#yourusername"  //username 需要拼接实例ID和用户名
  password: "yourpassword"
```

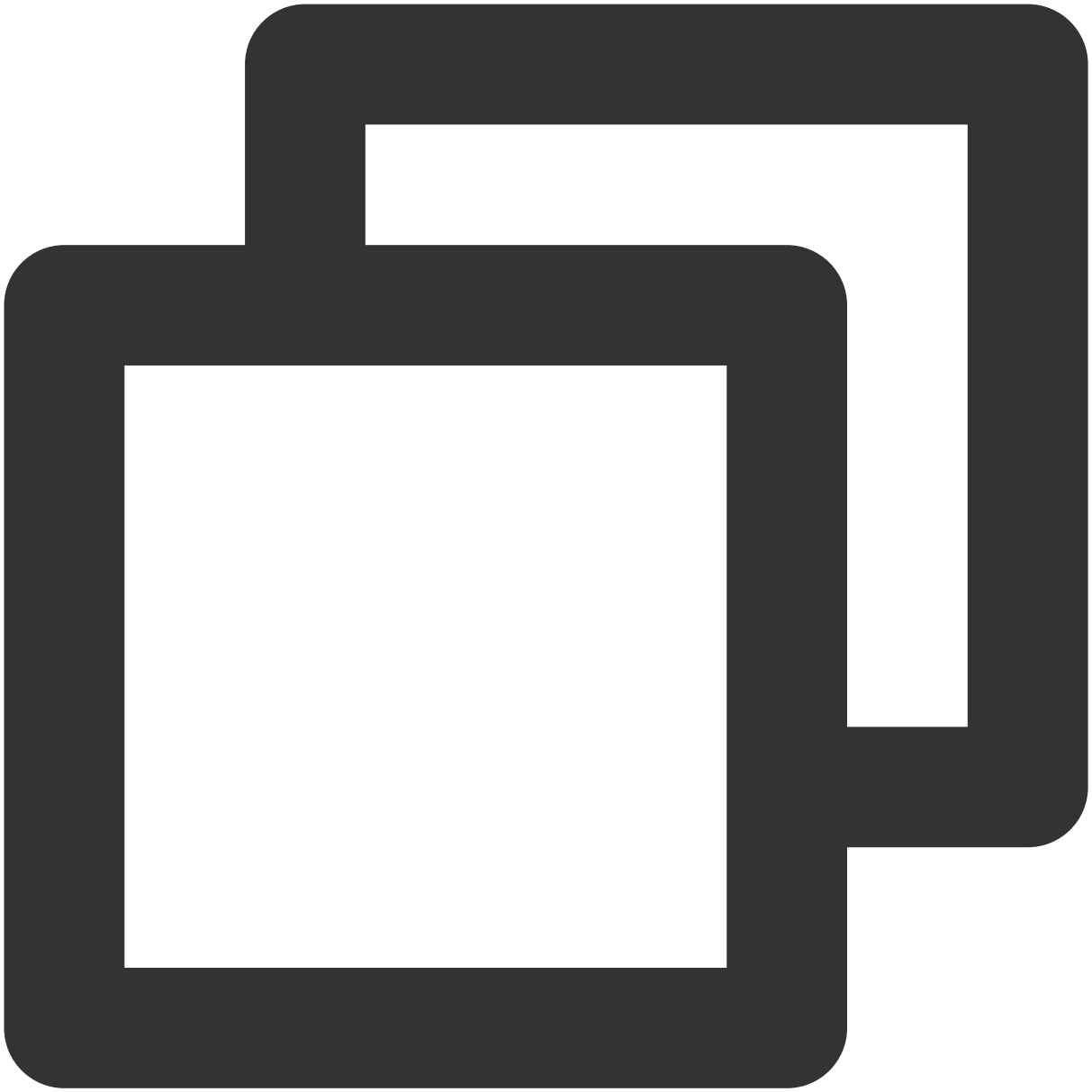
步骤4：Filebeat 发送消息

1. 执行如下命令启动客户端。



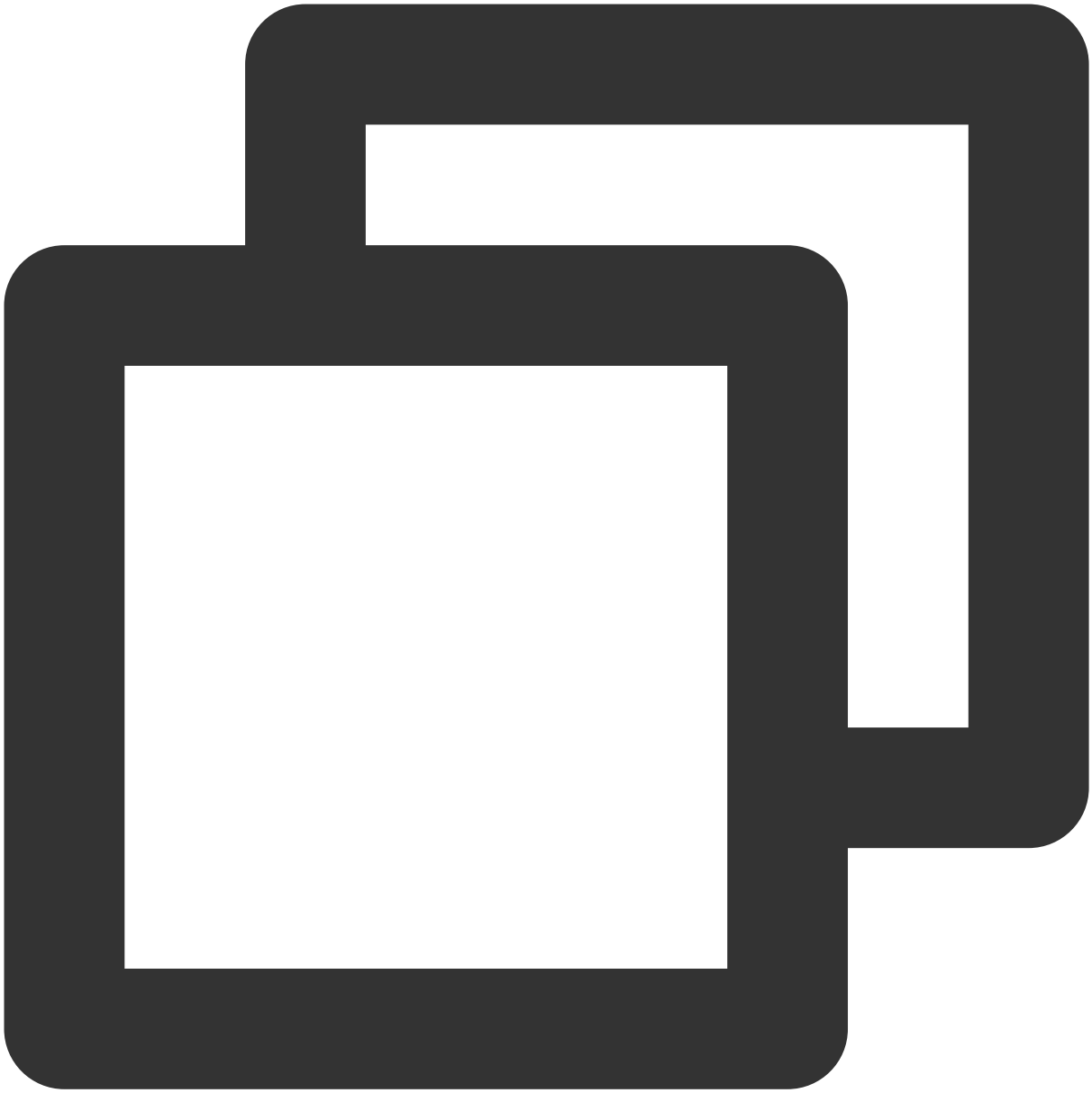
```
sudo ./filebeat -e -c filebeat.yml
```

2. 为监控文件增加数据（示例为写入监听的 testlog 文件）。



```
echo ckafka1 >> testlog
echo ckafka2 >> testlog
echo ckafka3 >> testlog
```

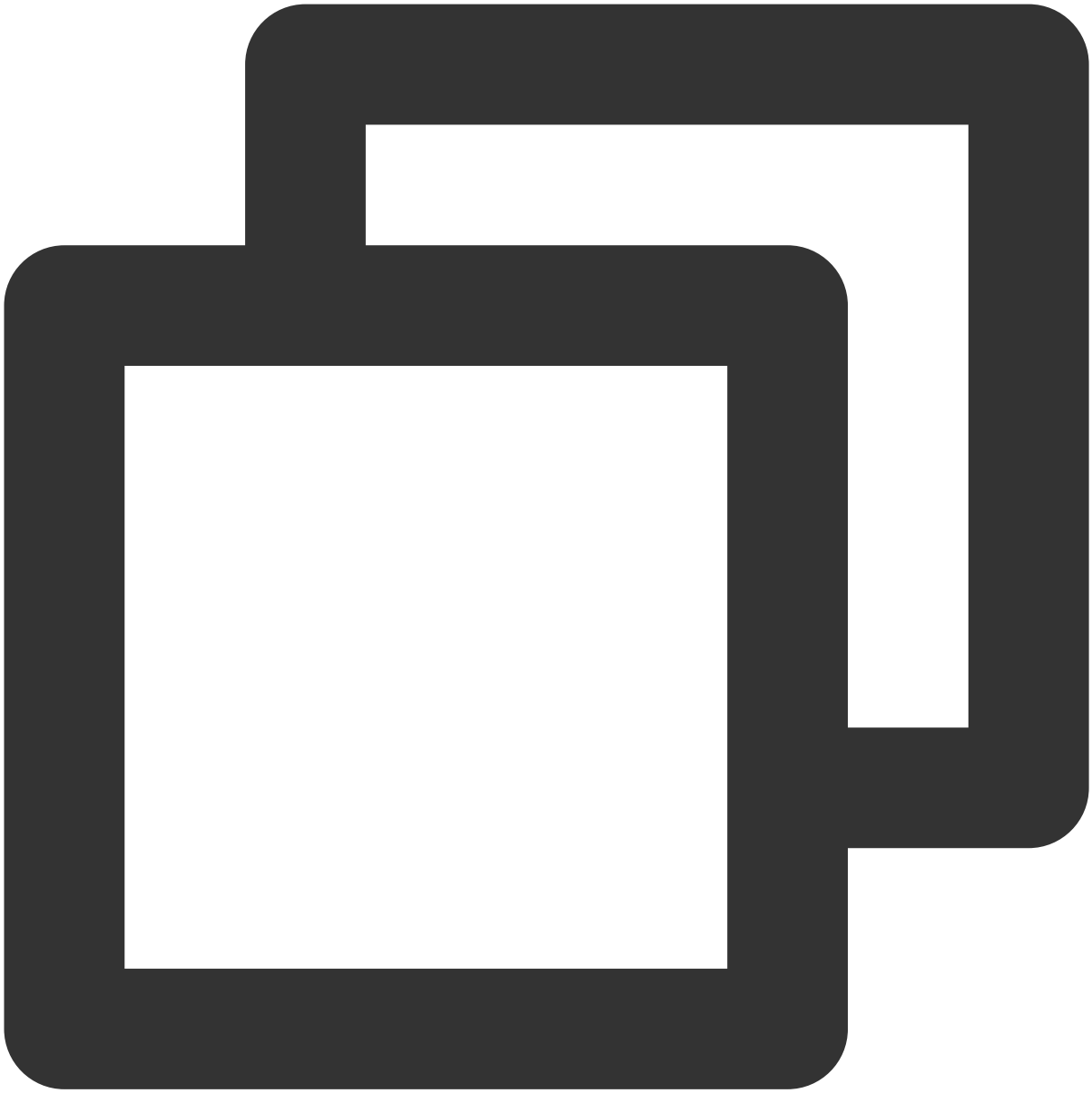
3. 开启 Consumer 消费对应的 Topic，获得以下数据。



```
{"@timestamp":"2017-09-29T10:01:27.936Z","beat":{"hostname":"10.193.9.26","name":"1
{"@timestamp":"2017-09-29T10:01:30.936Z","beat":{"hostname":"10.193.9.26","name":"1
{"@timestamp":"2017-09-29T10:01:33.937Z","beat":{"hostname":"10.193.9.26","name":"1
```

SASL/PLAINTEXT 模式

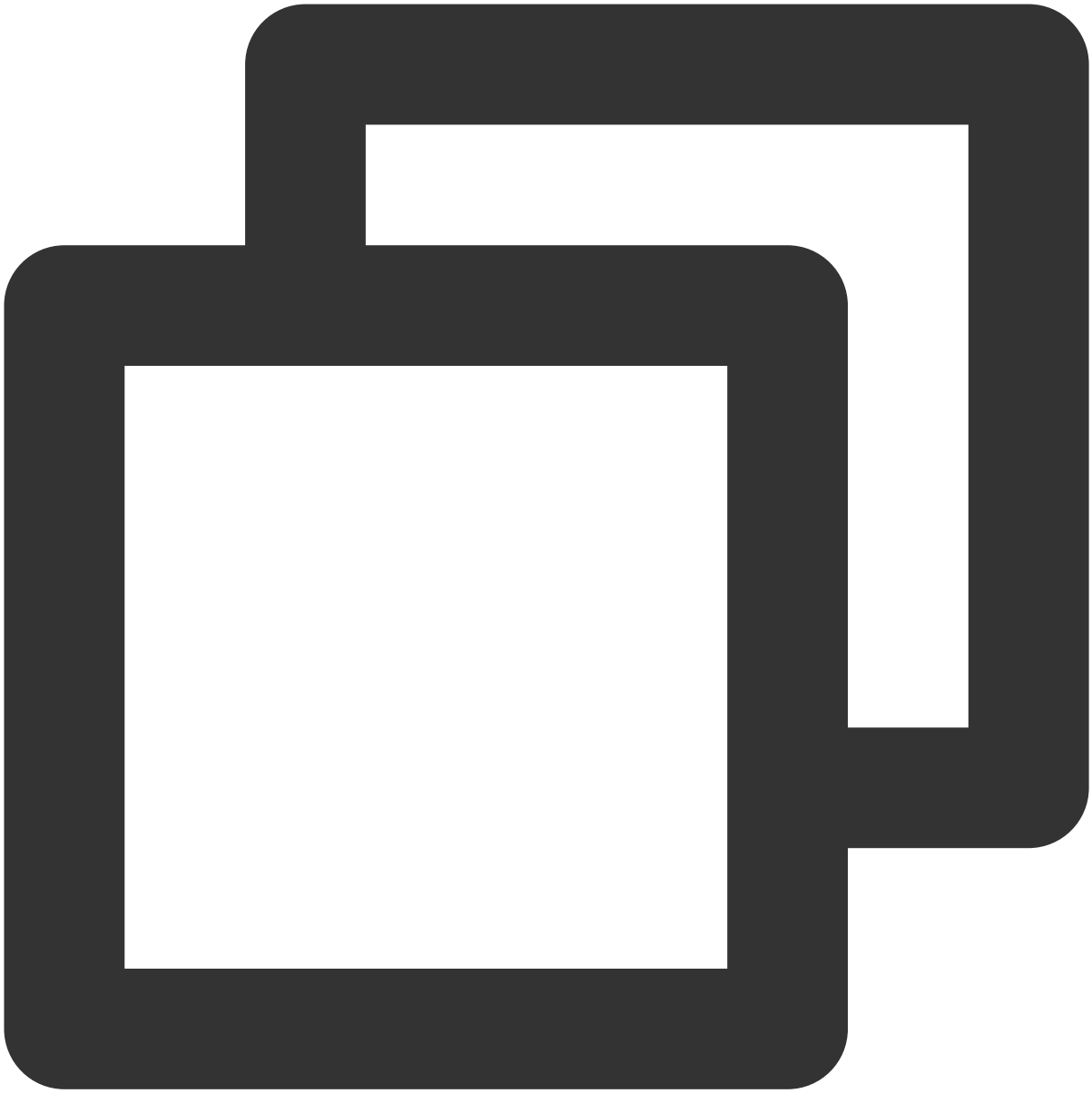
如果您需要进行 SALS/PLAINTEXT 配置，则需要配置用户名与密码。 在 Kafka 配置区域新增加 username 和 password 配置即可。



```
# SASL 需要配置下列信息，如果不需要则下面两个选项可不配置
 username: "yourinstance#yourusername"  //username 需要拼接实例ID和用户名
 password: "yourpassword"
```

常见问题

在 Filebeat 日志（默认路径 `/var/log/filebeat/filebeat` ）中，发现有大量 INFO 日志，例如：



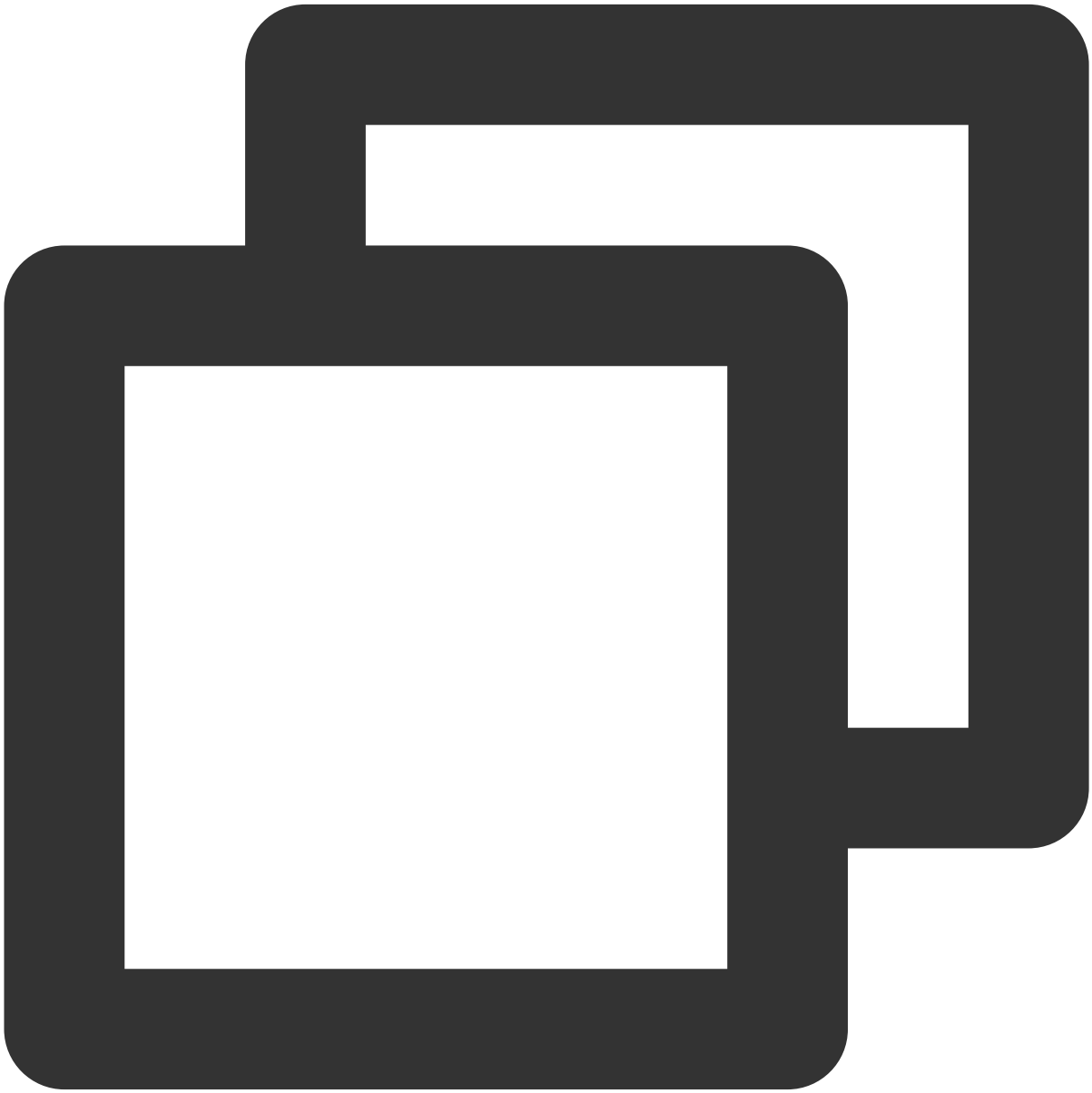
```
2019-03-20T08:55:02.198+0800    INFO    kafka/log.go:53 producer/broker/544 startin
2019-03-20T08:55:02.198+0800    INFO    kafka/log.go:53 producer/broker/544 state c
2019-03-20T08:55:02.198+0800    INFO    kafka/log.go:53 producer/leader/wp-news-fil
2019-03-20T08:55:02.198+0800    INFO    kafka/log.go:53 producer/broker/478 state c
2019-03-20T08:55:02.199+0800    INFO    kafka/log.go:53 Closed connection to broker
2019-03-20T08:55:02.199+0800    INFO    kafka/log.go:53 producer/leader/wp-news-fil
2019-03-20T08:55:02.199+0800    INFO    kafka/log.go:53 producer/leader/wp-news-fil
2019-03-20T08:55:02.199+0800    INFO    kafka/log.go:53 producer/leader/wp-news-fil
2019-03-20T08:55:02.199+0800    INFO    kafka/log.go:53 producer/leader/wp-news-fil
2019-03-20T08:55:02.199+0800    INFO    kafka/log.go:53 producer/leader/wp-news-fil
2019-03-20T08:55:02.199+0800    INFO    kafka/log.go:53 producer/leader/wp-news-fil
```

```
2019-03-20T08:55:02.199+0800    INFO    kafka/log.go:53 producer/broker/478 shut do
```

出现大量 INFO 可能是 Filebeat 版本有问题，因为 Elastic 家族的产品发版速度很频繁，而且不同大版本有很多不兼容。

例如：6.5.x 默认支持 Kafka 的版本是 0.9、0.10、1.1.0、2.0.0，而 5.6.x 默认支持的是0.8.2.0。

您需要检查配置文件中的版本配置：



```
output.kafka:
  version:0.10.2 // 根据不同 CKafka 实例开源版本配置
```

说明与注意

发送数据到 CKafka，不能设置压缩 compression.codec。
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。
使用 LZ4 压缩方法时，程序不能正常运行，可能的原因如下：
消息格式错误。CKafka 默认版本为0.10.2，您需要使用 V1 版本的消息格式。
不同 Kafka 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。

跨可用区部署

最近更新时间：2024-01-09 14:56:36

CKafka 跨可用区部署

CKafka 专业版支持跨可用区部署，在拥有3个或3个以上可用区的地域购买 CKafka 实例时，可以最多选择四个可用区购买跨可用区实例。该实例分区副本会强制分布在各个可用区节点上，这种部署方式能够让您的实例在单个可用区不可用情况下仍能正常提供服务。

说明：

仅专业版支持跨可用区部署，标准版无法支持。

CKafka 跨可用区部署原理

CKafka 的跨可用区部署分为网络层、数据层和控制层。

网络层

CKafka 会为客户端暴露一个 VIP，客户端在连接到 VIP 后，会拿到主题分区的元数据信息（该元数据通常是地址会通过同一个 VIP 的不同 port 进行一一映射）。

是一个可以随时 failover 到另一个可用区的 VIP，当某个可用区不可用时，该 VIP 会自动漂移到该地域另一个可用的节点，从而实现跨可用区容灾。

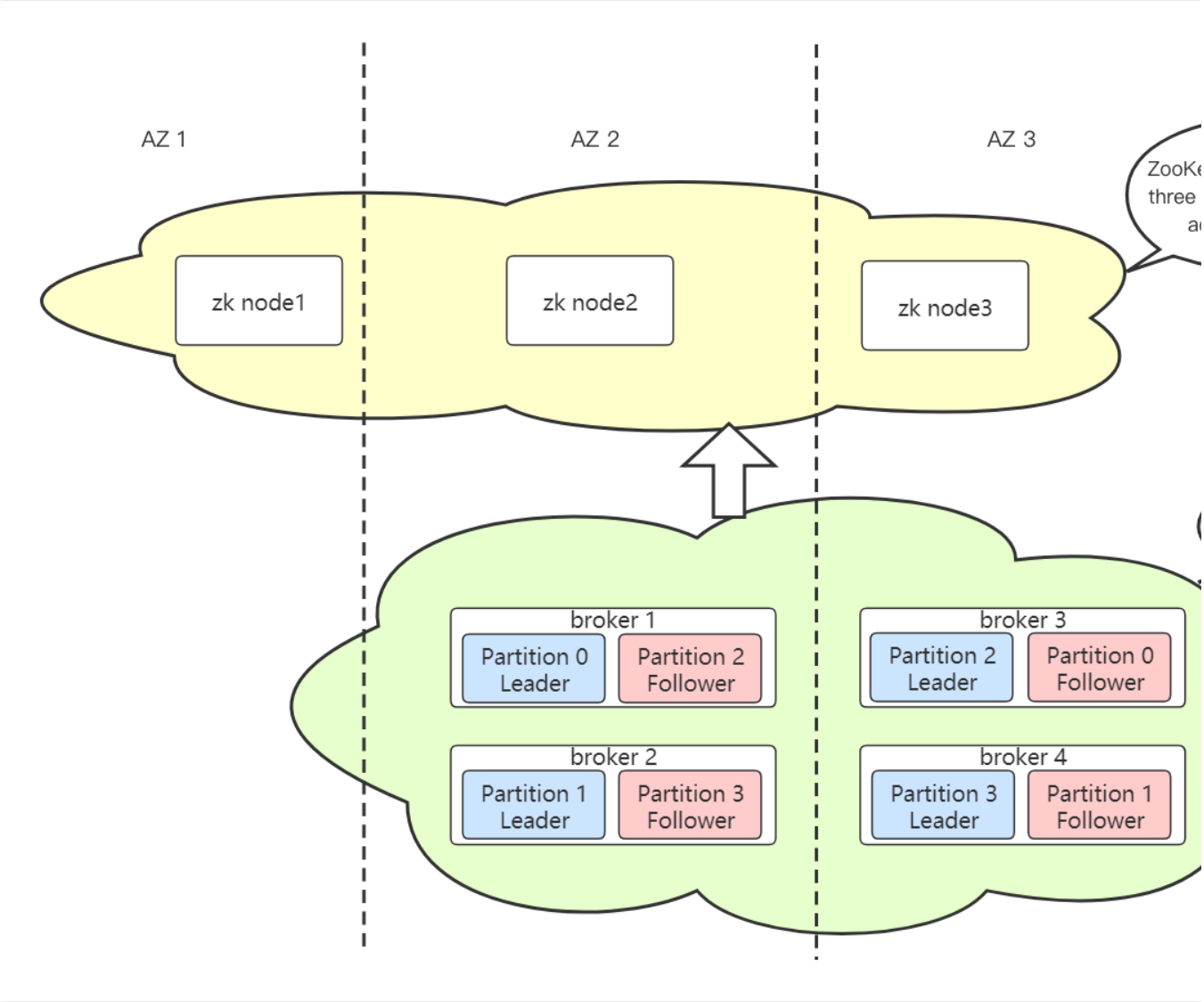
数据层

CKafka 数据层和原生 Kafka 采用相同的分布式部署方式，即多个数据副本分布在不同 broker 节点，不同节点会部署在不同可用区。在处理某个分区时，不同的节点之间会有 leader-follower 的关系，当 leader 发生异常不在线时，集群控制节点（Controller）会选举出新的分区 leader 来承接这个分区的请求。

对于客户端来说，当某个可用区出现异常不可用后，如果某个主题分区的 leader 位于不可用区 broker 节点上，则原先建立的相关链接会出现超时或者链接被关闭的情况，当该分区 leader 节点异常之后，Controller（若 Controller 节点异常，剩余节点会竞选出新的 Controller 节点）会选举出新的leader节点提供服务，leader 切换 时间在秒级（具体切换时间与集群节点个数，元数据大小成正比）。客户端会定时刷新主题分区元数据信息，链接新的 leader 节点进行生产消费。

控制层

CKafka 的控制层和原生 Kafka 采用相同的技术方案，依赖 zookeeper 对 broker 节点进行服务发现和集群 Controller 选举。**支持跨可用区部署的CKafka实例，其 zookeeper 集群中 zk 节点（以下简称 zk 节点）部署在三个可用区（或机房）。**当其中任意一个可用区的 zk 节点出现故障断连，整个 zk 集群仍可以正常提供服务。



跨可用区部署优劣势

优势

可以大幅度提升集群的容灾能力，当单个可用区出现意外的网络不稳定、断电重启等不可抗力风险时，仍能保证客户端在短时间等待重连后恢复消息的生产和消费。

劣势

如果采取跨可用区部署，由于分区副本分布在多个可用区上，故消息复制相比单个可用区存在额外的跨区网络时延，该时延会直接影响到生产（客户端 Ack 参数大于1，或者等于-1, all）的客户端写入耗时。目前广州、上海、北京几个主要地域跨可用区的时延一般10ms~40ms。

跨可用区部署场景解析

单 AZ 不可用

单个 AZ 不可用后，如前文对原理的解析，客户端会出现断连重连，重连后服务仍能正常提供。
由于管控 API 服务目前不支持跨可用区部署，所以在单个 AZ 不可用之后，可能出现无法通过控制台创建 Topic，配置 ACL 策略，查看监控等现象，但不会影响存量业务的生产消费

两个 AZ 网络隔离

如果两个 AZ 之间出现网络隔离，即无法相互通信，则可能会出现集群脑裂的现象，即两个可用区的节点都提供服务，但其中一个可用区的数据写入会在集群恢复后视为脏数据。
考虑如下场景，当集群 Controller 节点和 zk 集群一个 zk 节点与其他节点发生网络隔离。此时，其他节点会重新竞选产生新的 Controller（因为 zk 集群多数节点网络通信正常，则 Controller 可以竞选成功）但是发生网络隔离的 Controller 仍然认为自己是 Controller 节点，这时候集群会出现脑裂的情况。
此时客户端的写入需要分情况考虑，这里举个例子，当客户端的 Ack 策略等于 -1 或者 all，副本数为2时，假设集群是3节点，脑裂之后会2:1分布，原先 leader 在1节点地域的分区写入会报错，另一边则会正常写入。而一旦是3副本且配置了 Ack = -1 或者 all，则两边都不会写入成功。这时就需要根据具体参数配置来确定进一步的处理方案。
在集群网络恢复后，客户端无需做操作即可恢复生产消费，但是由于服务端会重新对数据进行归一化，其中一个分裂节点的数据会被直接截断，但对于多副本跨区的数据存储方式来说，这种截断也并不会带来数据丢失。

操作步骤

购买实例选择多个可用区

1. 登录 CKafka 控制台。
2. 在左侧导航栏单击**实例列表**，单击**新建**进入实例购买页。
3. 在实例购买页，根据自身业务需求选择购买信息。
计费模式：包年包月
规格类型：根据自身业务需求选择标准版或者专业版。
Kafka 版本：根据您的业务需求选择 Kafka 版本，可参见 CKafka 版本选择建议。
地域：选择和部署客户端的资源相近的地域。
可用区：根据实际需要选择可用区。
标准版：不支持多可用区部署。
专业版：若当前地域支持多可用区部署，则最多可选择4个可用区进行部署。关于跨可用区部署原理介绍请参见 跨可用区部署。
产品规格：根据峰值带宽和磁盘容量选择对应的型号。
消息保留：范围在24小时 - 2160小时。
在磁盘容量不足（即磁盘水位达到90%）时，将会提前删除旧的消息，以确保服务可用性。

私有网络：若用户需要接入其他私有网络可参见 添加路由策略 修改路由接入规则。

标签：选填，具体使用方法可参见 标签管理。

实例名称：购买多个实例时，支持创建实例后缀数字自动升序以及指定模式串功能。具体操作参见 批量连续命名或指定模式串命名。

4. 单击**立即购买**，完成实例创建。

日志接入
日志服务 CLS

最近更新时间：2024-01-09 14:56:36

简介

您可以将日志主题的数据投递到腾讯云 Ckafka，然后用于您的实时流计算、或者入库等场景。如果您没有购买腾讯云 Ckafka 实例，可以考虑使用日志服务（Cloud Log Service，CLS）自带的 Kafka 协议消费功能。

前提条件

已开通腾讯云消息队列 （CKafka）。
确保当前操作账号拥有开通投递到 Ckafka 的权限。如果您是子账号，一般需要主账号进行相关授权，方可使用。权限问题请参见 自定义权限策略示例。

操作步骤

1. 在日志主题同地域下，创建一个 Ckafka 实例。详情请参见 创建实例。
2. 在日志主题同地域下，根据如下配置参数，创建一个 Topic。详情请参见 创建 Topic。
预设 ACL 策略：关闭预设 ACL 策略。
展示高级配置：
CleanUp.policy：选择 **delete**。该参数需设置为 delete，否则会投递失败。
max.message.bytes：设置为 ≥ 8MB。该参数若小于8MB，会因 CLS 侧的单条 message 过大，无法写入 Ckafka Topic，导致投递失败。
3. 前往 日志服务控制台，并按需选择不同的操作，进入投递任务管理页面或者日志主题管理页面。
在左侧导航栏中，单击**投递任务管理**，选择地域、日志集和日志主题。
在左侧导航栏中，单击**日志主题**，选择需要配置投递到 Ckafka 任务的日志主题，进入日志主题管理页面。
4. 单击**投递到 Ckafka** 页签，进入投递到 Ckafka 配置页面。
5. 单击右侧的**编辑**，开启投递到 Ckafka 开关，选择相应的 Ckafka 实例以及对应的 Topic，选择需要投递的日志字段。
6. 单击**确定**，启动投递到 Ckafka，任务状态显示为“已开启”则表示开启成功。
注意：
如需在投递至 Ckafka 前对日志进行清洗加工过滤，可参考使用 数据加工 操作。

常见问题

日志数据无法投递 Ckafka，怎么办？

如果您在腾讯云 Ckafka 开启了 ACL 鉴权，会导致日志数据无法投递。请关闭该 Topic 的 ACL。

提示没有读写 Ckafka Topic 的权限，怎么办？

如果您直接使用 API 接口投递数据到 Ckafka，可能会存在读写 Ckafka Topic 的权限问题。因为，如果您在控制台使用该功能，系统会引导您完成相关授权，如果您直接调用 API 投递，则需要手动授权。具体的排查和解决方案请参见 投递权限查看及配置。

替换支撑路由（旧）

最近更新时间：2024-01-09 14:56:36

背景

为了给您提供更稳定可靠的服务，建议切换为新的支撑路由。

影响

因为需要更新生产、消费端的 bootstrap-server 地址，并且重启服务， 所以这里会影响业务的生产、消费短暂性中断。

操作步骤

1. 新建路由：新建一条支撑路由。
2. 切换路由：切换所有生产端、消费端的 CKafka 链接地址 bootstrap-server 为新创建的支撑环境路由；(生产端、消费端切换不分先后顺序，只要保证全部切换完)。
3. 重启服务：重启生产、消费端(生产、消费重启顺序无关，根据业务情况自行决策)。
4. 验证业务：持续观察一段时间业务是否稳定, 建议持续观察3小时以上。
5. 删除路由：删除旧支撑路由。

回滚

验证业务阶段 如果发现异常，则需要回滚，回滚的前提是旧支撑路由未删除 具体回滚操作：
1. 所有生产、消费端替换旧支撑路由的地址。
2. 重启所有生产、消费端服务。
3. 验证业务。