消息队列 CKafka 版

通用参考

产品文档

腾讯云

文档目录

通用参考
对 CKafka 进行生产和消费压力测试

最近更新时间：2024-01-09 15:02:47

测试工具

Kafka Producer 和 Consumer 的性能测试均可使用 Kafka 客户端自带的开源脚本，主要输出每秒发送消息量（MB/second）和每秒发送消息数（records/second）两项指标。

Kafka Producer 测试脚本：`$KAFKA_HOME/bin/kafka-producer-perf-test.sh`

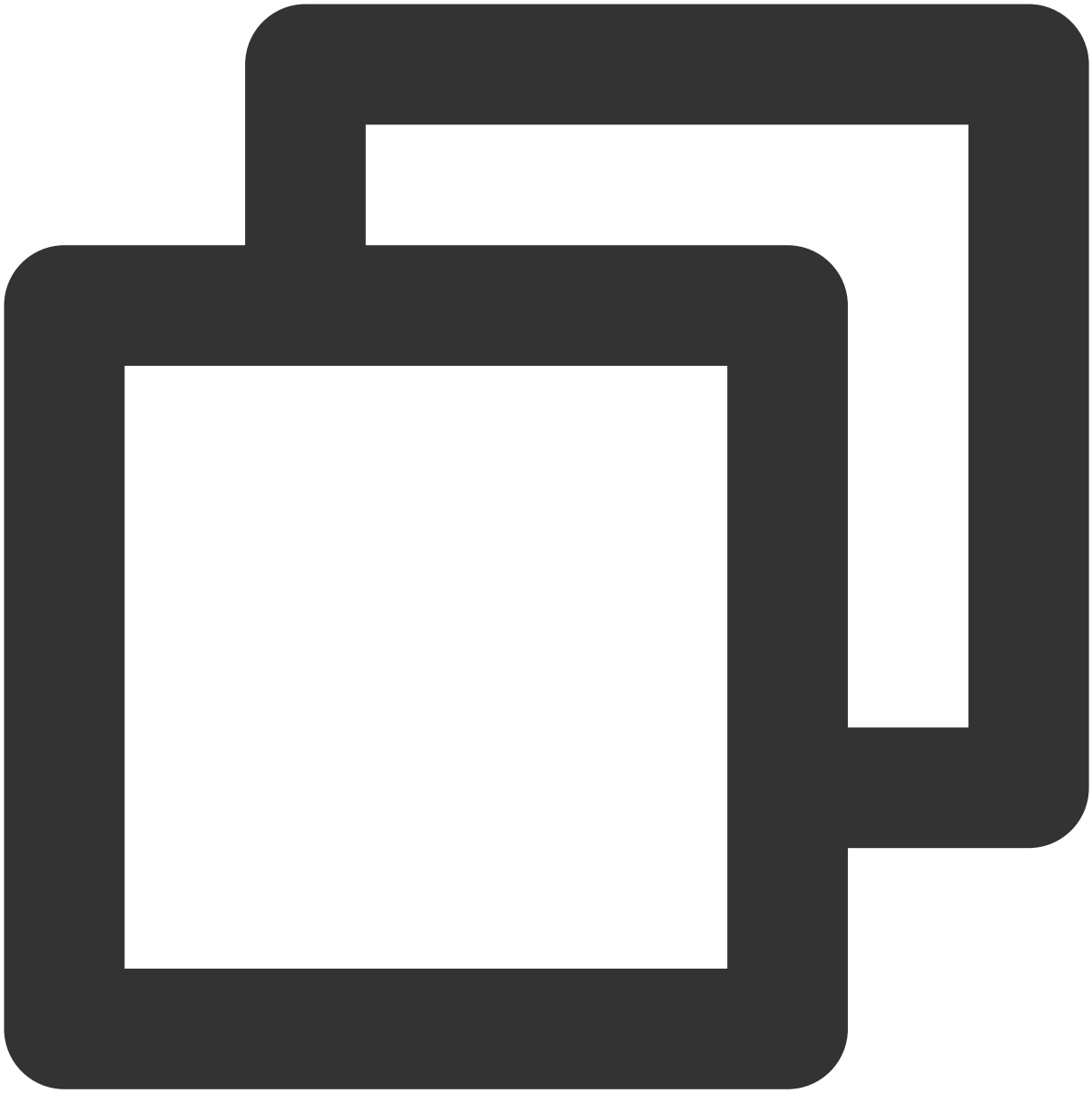
Kafka Consumer 测试脚本：`$KAFKA_HOME/bin/kafka-consumer-perf-test.sh`

测试命令

说明：

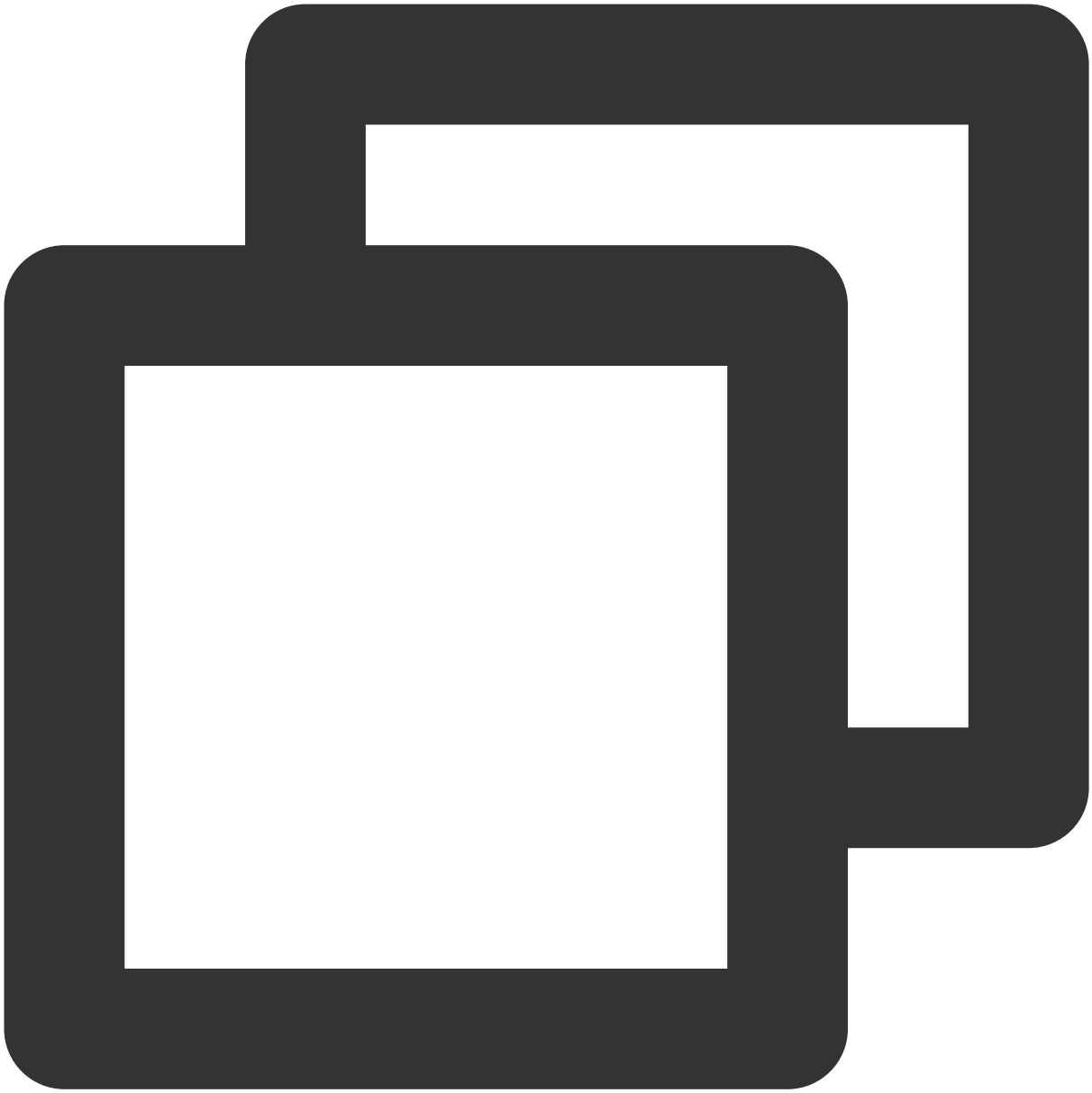
以下命令中的 `ckafka vip:vport` 应替换为您实际实例分配的 IP 和端口。

生产测试命令示例：



```
bin/kafka-producer-perf-test.sh
--topic test
--num-records 123
--record-size 1000
--producer-props bootstrap.servers= ckafka vip : port
--throughput 20000
```

消费测试命令示例：



```
bin/kafka-consumer-perf-test.sh
--topic test
--new-consumer
--fetch-size 10000
--messages 1000
--broker-list bootstrap.servers=ckafka vip : port
```

测试建议

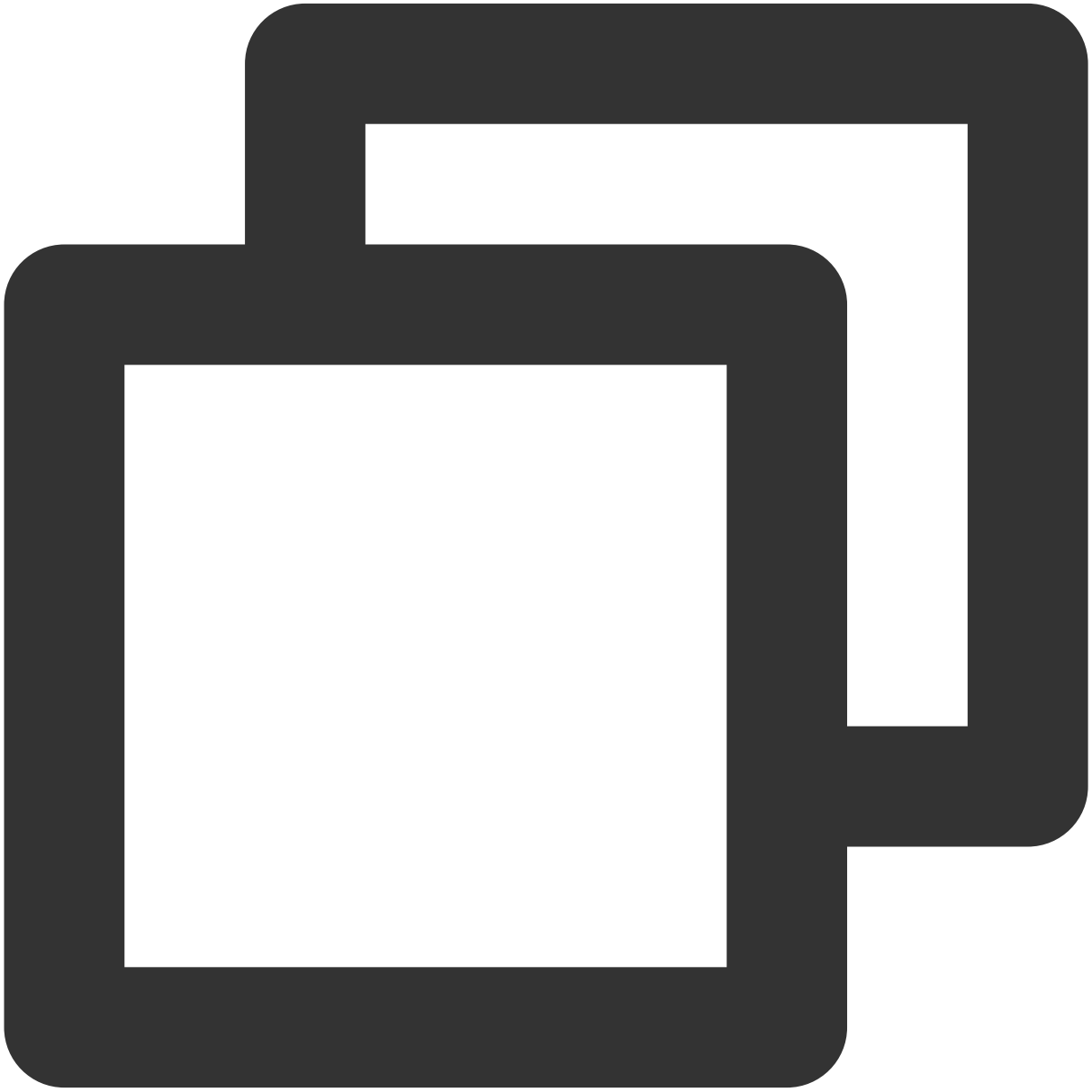
为了提高吞吐量，建议创建分区时数量 ≥ 3 （因后端 CKafka 集群节点数量最少是3，如只创建1个分区则分区会分布在一个 Broker 上面，影响性能）。
由于 CKafka 是分区级别消息有序的，因此过多的分区也会影响生产性能，根据实际压测，建议分区数不超过6。
为了保证压力测试的效果，需要多客户端模拟一定的并发，建议采用多台机器作为压测客户端（生产端），每台启动多个压测程序，提高并发。此外建议每1s启动一个生产者，避免同时启动所有生产者导致测试机器高负载。

常见参数配置说明

最近更新时间：2024-02-20 15:13:35

Broker 配置参数说明

当前 CKafka broker 端的一些配置如下，供参考：



```
# 消息体的最大 大小，单位是字节
```

```
message.max.bytes=1000012

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

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

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

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

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

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

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

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

说明：

其他未列出的 Broker 配置参考 开源 Kafka 默认配置。

Topic 配置参数说明

1. 选取合适的分区数量

从生产者的角度来看，向不同的 partition 写入是完全并行的；从消费者的角度来看，并发数完全取决于 partition 的数量（如果 consumer 数量大于 partition 数量，则必有 consumer 闲置）。因此选取合适的分区数量对于发挥 CKafka 实例的性能十分重要。

partition 的数量需要根据生产和消费的吞吐来判断。理想情况下，可以通过如下公式来判断分区的数目：

说明：

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

其中，Num 代表 partition 数量，T 代表目标吞吐量，PT 代表生产者写入单个 partition 的最大吞吐，CT 代表消费者从单个 partition 消费的最大吞吐。则 partition 数量应该等于 T/PT 和 T/CT 中较大的那一个。

在实际情况中，生产者写入单个 partition 的最大吞吐 PT 的影响因素和批处理的规模、压缩算法、确认机制、副本数等有关。消费者从单个 partition 消费的最大吞吐 CT 的影响因素和业务逻辑有关，需要在不同场景下实测得出。

通常建议 partition 的数量一定要大于等于消费者的数量来实现最大并发。 如果消费者数量是 5，则 partition 的数目也应该是 ≥ 5 的。同时，过多的分区会导致生产吞吐的降低和选举耗时的增加，因此也不建议过多分区。提供如下信息供参考：

单个 partition 是可以实现消息的顺序写入的。

单个 partition 只能被同消费者组的单个消费者进程消费。

单个消费者进程可同时消费多个 partition，即 partition 限制了消费端的并发能力。

partition 越多则失败后 leader 选举的耗时越长。

offset 的粒度最细是在 partition 级别的，partition 越多，查询 offset 就越耗时。

partition 的数量是可以动态增加的，只能增加不能减少。但增加会出现消息 rebalance 的情况。

2. 选取合适的副本

目前为了保证可用性副本数必须大于等于2，如果需要保障高可靠建议3副本。

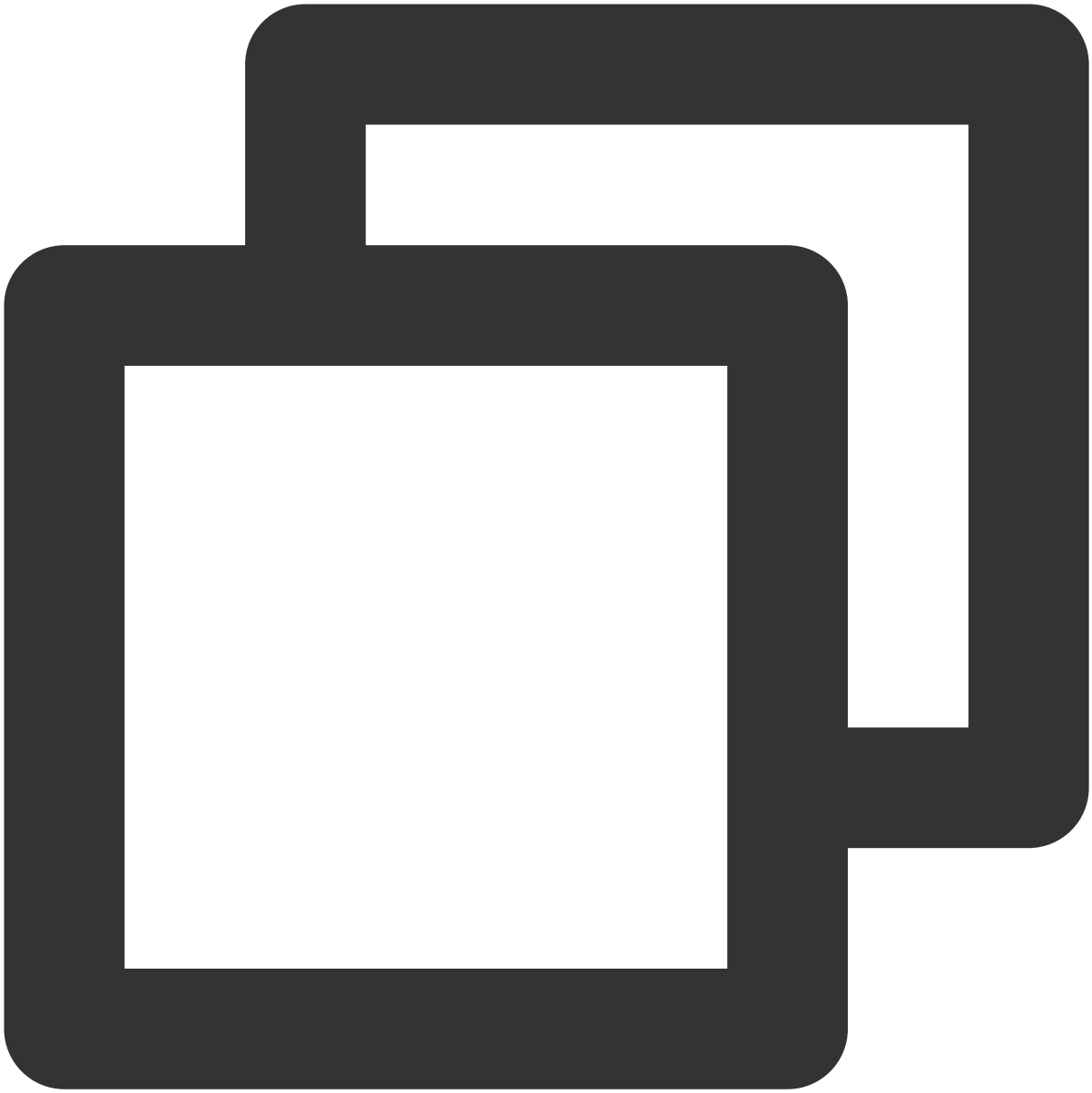
注意：

副本数会影响生产/消费流量，如3副本则实际流量 = 生产流量 × 3。

3. 日志保留时间

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

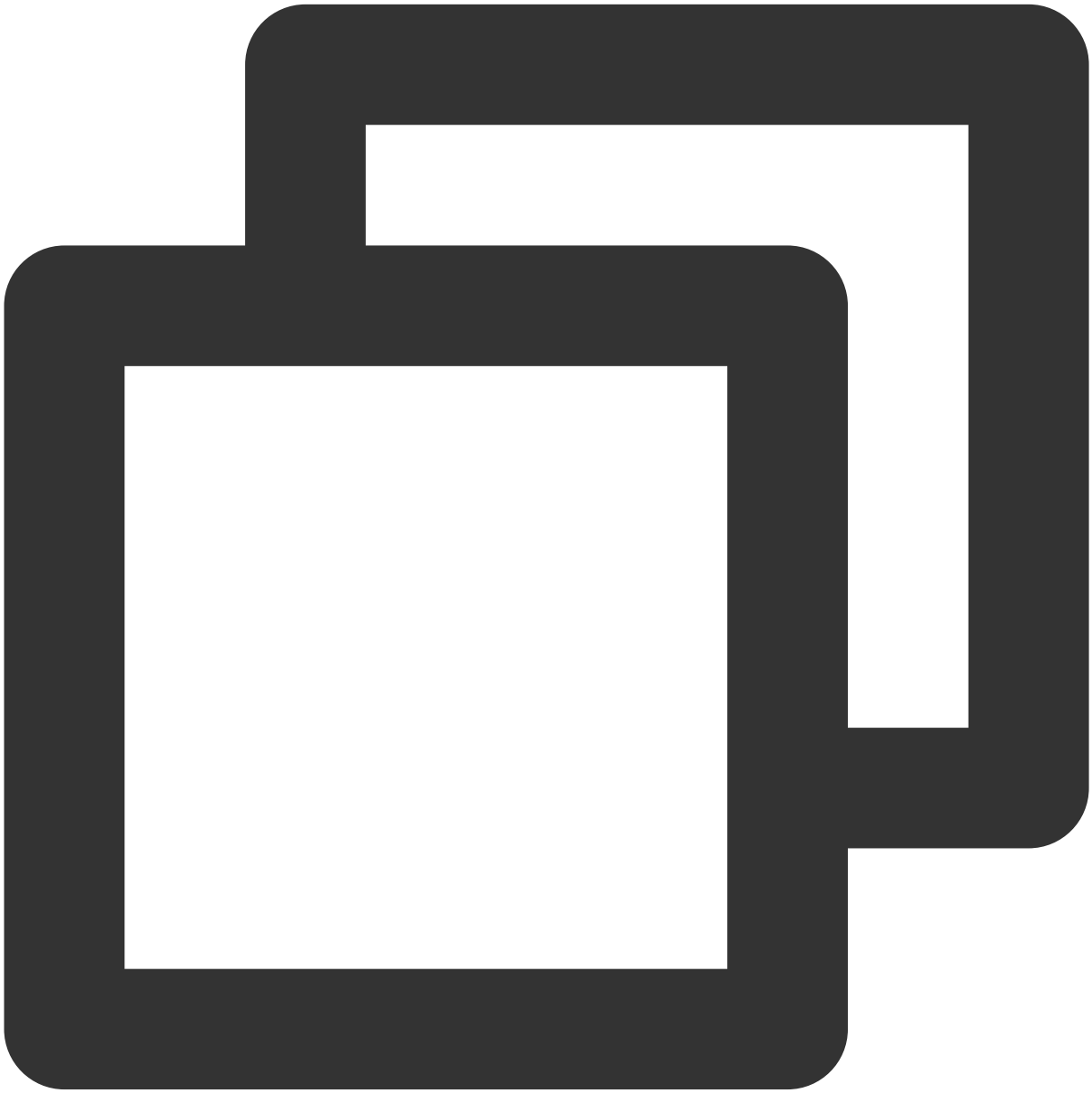
4. 其他 Topic 级别配置说明



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

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

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

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

生产者配置指南

生产端常用参数配置如下，建议客户根据实际业务场景调整配置：

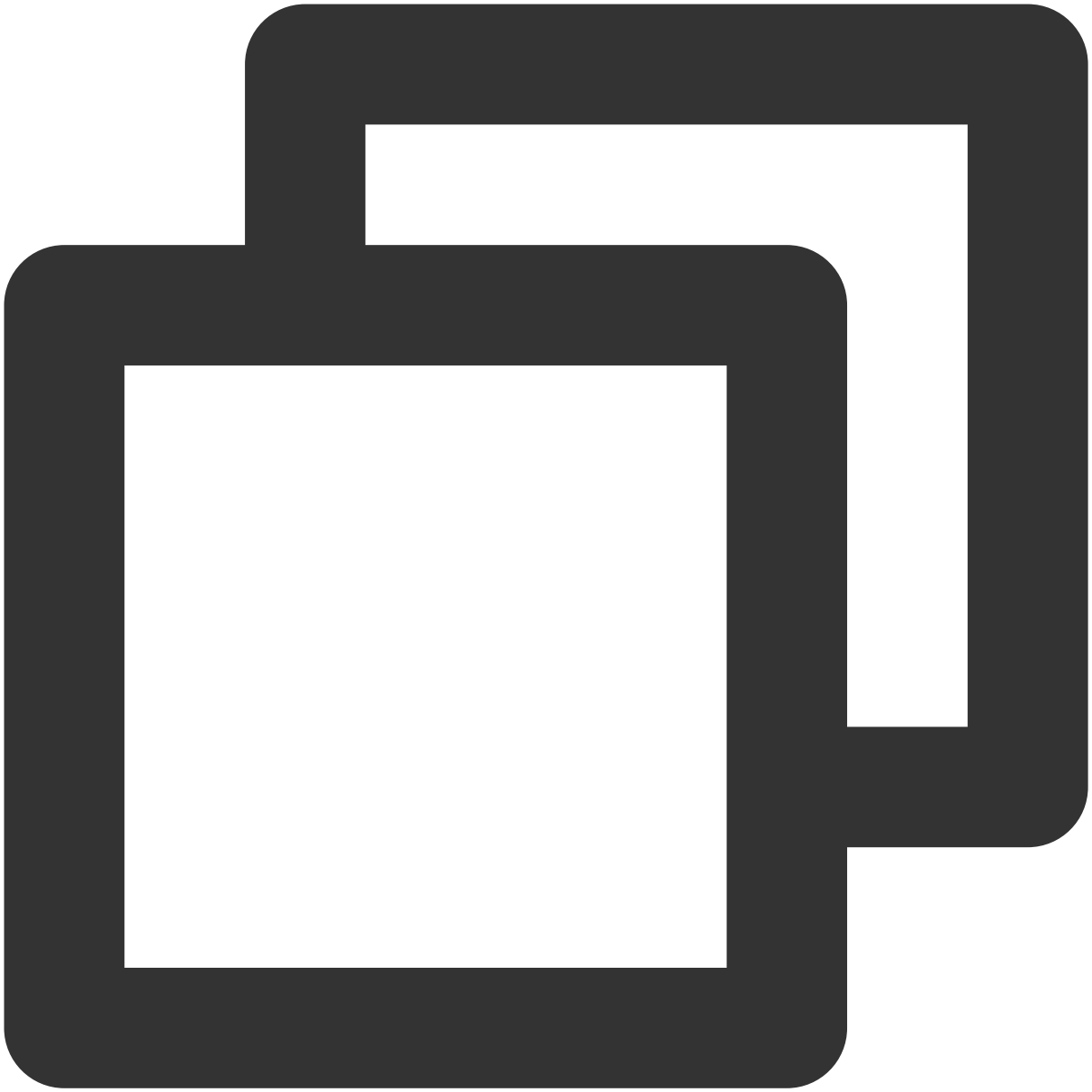


```
# 生产者会尝试将业务发送到相同的 Partition 的消息合包发送到 Broker，batch.size 设置合包的大小
batch.size=16384
```

```
# Kafka producer 的 ack 有 3 种机制，分别说明如下：
# -1 或 all：Broker 在 leader 收到数据并同步给所有 ISR 中的 follower 后，才应答给 Producer
# 0：生产者不等待来自 broker 同步完成的确认，继续发送下一条（批）消息。这种配置生产性能最高，但数
# 1：生产者在 leader 已成功收到的数据并得到确认后再发送下一条（批）消息。这种配置是在生产吞吐和数

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

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

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

# 当生产消息的速度比 Sender 线程发送到 Broker 速度快，导致 buffer.memory 配置的内存用完时会阻
max.block.ms=60000

# 设置消息延迟发送的时间，这样可以等待更多的消息组成 batch 发送。默认为0表示立即发送。当待发送的消
linger.ms=0

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

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

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

# 客户端在每个连接上最多可发送的最大的未确认请求数，该参数大于1且 retries 大于0时可能导致数据乱序
max.in.flight.requests.per.connection=5


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

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

消费者配置指南

消费端常用参数配置如下，建议客户根据实际业务场景调整配置：



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

# 当 auto.commit.enable=true 时，自动提交 Offset 的时间间隔，建议设置至少1000
auto.commit.interval.ms=5000

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

```
# 标识消费者所属的消费分组
group.id=""

# 使用 Kafka 消费分组机制时，消费者超时时间。当 Broker 在该时间内没有收到消费者的心跳时，认为该消
session.timeout.ms=10000

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

# 使用 Kafka 消费分组机制时，再次调用 poll 允许的最大间隔。如果在该时间内没有再次调用 poll，则认
max.poll.interval.ms=300000

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

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

# Fetch 请求等待时间
fetch.max.wait.ms=500

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

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

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

接入低版本自建 Kafka

最近更新时间：2024-01-09 15:02:47

CKafka 兼容0.9及以上的生产/消费接口（目前可以直接购买的版本包括0.10.2、1.1.1、2.4.1、2.8.1、3.2.3版本），如果接入低版本（如0.8版本）的自建 Kafka，您需要对接口进行相应改造。本文将从生产端和消费端对比0.8版本 Kafka 和高版本 Kafka，并提供改造方式。

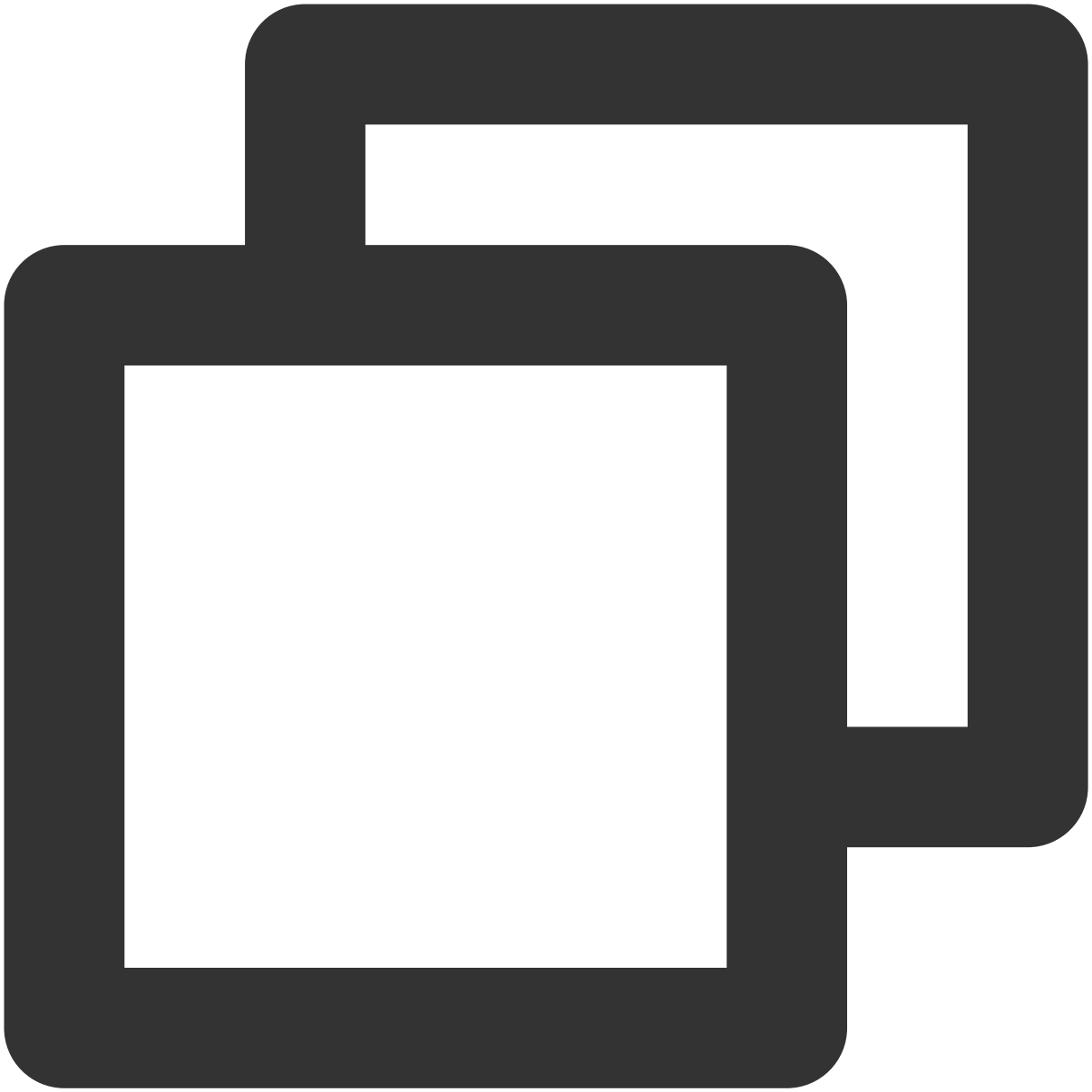
Kafka Producer

概述

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

新旧版本 Producer API 对比

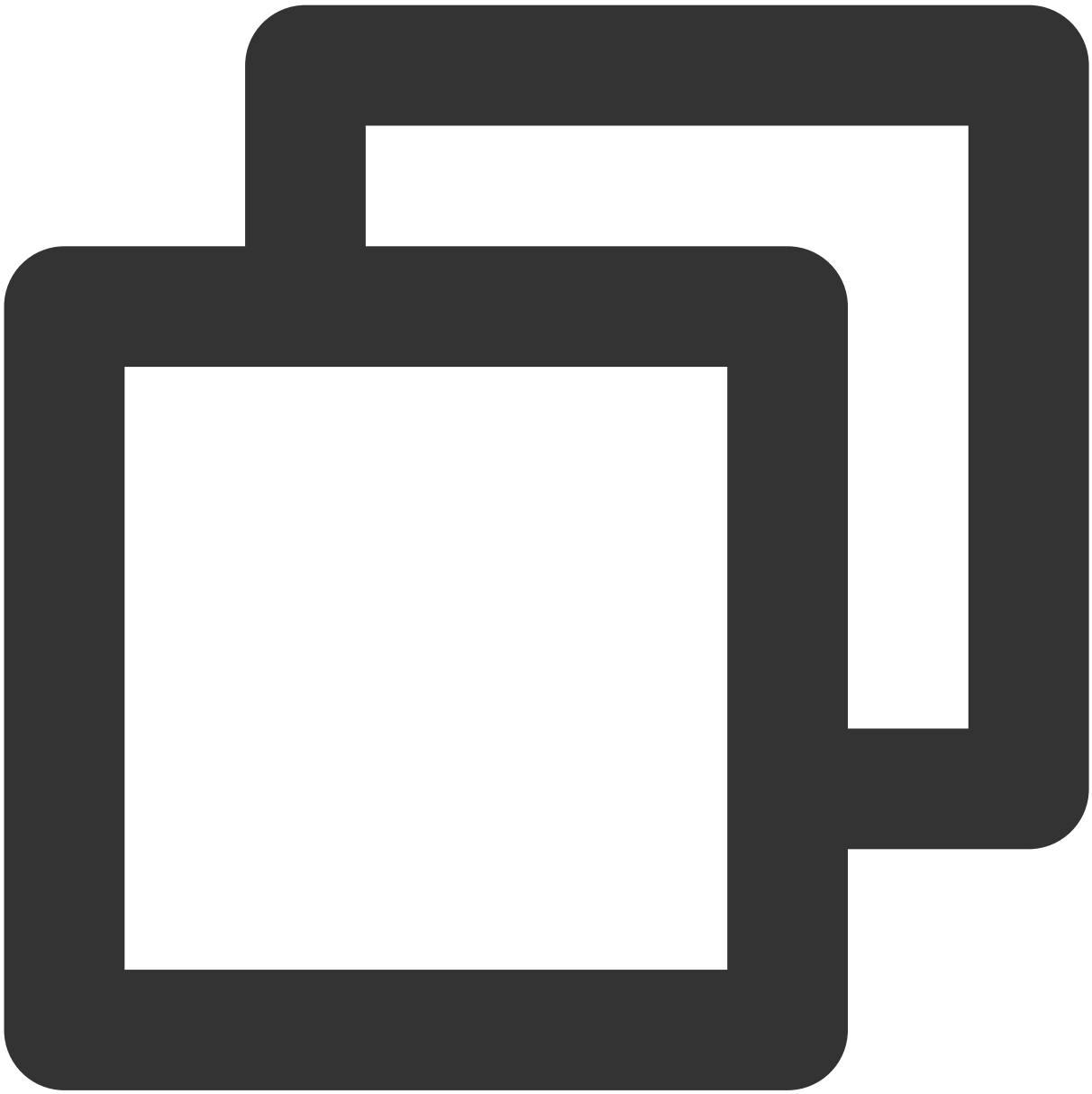
新版 Producer API Demo：



```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:4242");
props.put("acks", "all");
props.put("retries",0);
props.put("batch.size", 16384);
props.put("linger.ms", 1);
props.put("buffer.memory", 33554432);
props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer
props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializ
Producer<String, String> producer = new KafkaProducer<>(props);
producer.send(new ProducerRecord<String, String>("my-topic", Integer.toString(0), I
```

```
producer.close();
```

旧版 Producer API Demo：



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

```
producer.send(data);
producer.close();
```

可以看出新旧版本的使用方法基本一致，只有一些参数的配置不同，改造代价不大。

兼容性说明

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

Kafka Consumer

概述

开源 Apache Kafka 0.8版本中提供了两种消费者 API，分别为：
High Level Consumer API （屏蔽配置细节）
Simple Consumer API （配置细节支持指定）
Kafka 0.9.x 版本引入了 New Consumer，其融合了 Old Consumer（0.8版本）两种 Consumer API 的特性，减轻了 ZooKeeper 的负载。
因此下文给出了0.8版本 Consumer 转换为0.9版本 New Consumer 的方式。

新旧版本 Consumer API 对比

0.8版本 Consumer API

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

0.9版本 New Consumer API

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

优势：

Coordinator 引入：

当前版本的 High Level Consumer 存在 Herd Effect 和 Split Brain 的问题。将失败探测和 Rebalance 的逻辑放到一个高可用的中心 Coordinator，那么这两个问题即可解决。同时还可很大程度的减少 ZooKeeper 的负载。

允许自己分配 Partition：

为了保持本地每个分区的一些状态不变，所以需要将 Partition 的映射也保持不变。另外一些场景是为了让 Consumer 与地域相关的 Broker 关联。

允许自己管理 Offset：

可以根据自己需要去管理 Offset，实现重复、跳跃消费等语意。

Rebalance 后触发用户指定的回调。

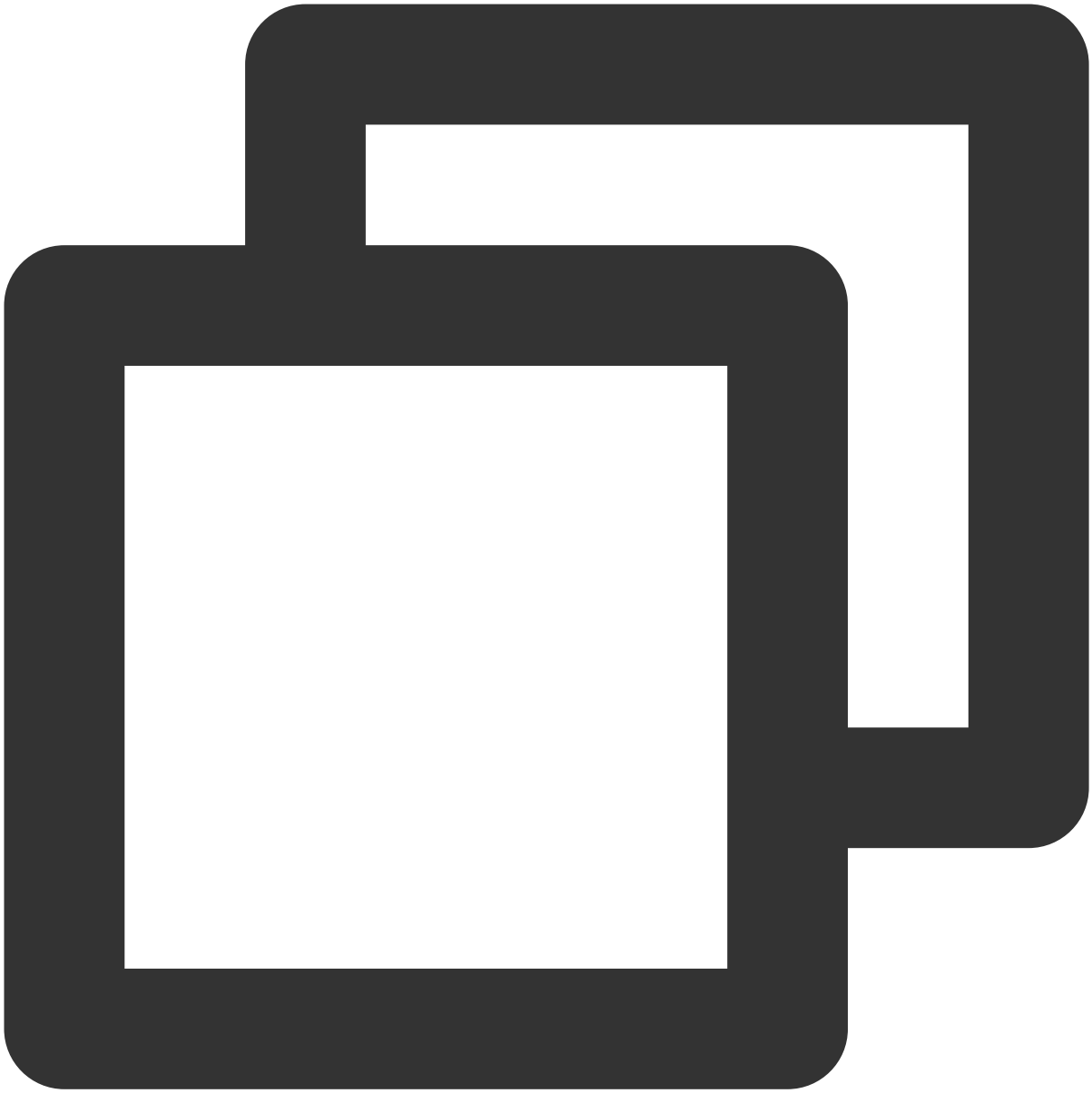
非阻塞式 Consumer API。

新旧版本 Consumer API 功能对比

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

Old Consumer 转换 New Consumer

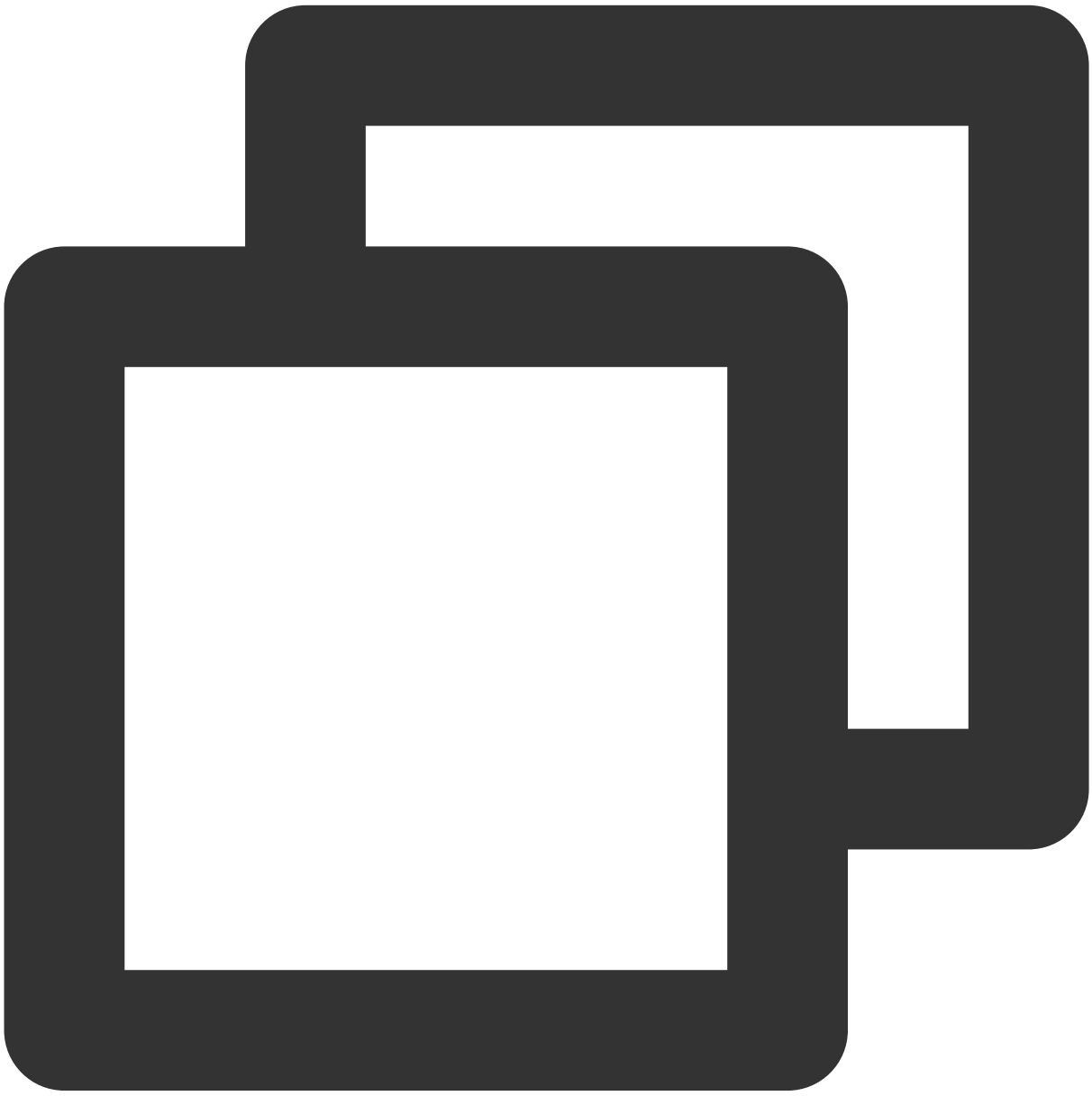
New Consumer：



```
//config中主要变化是 ZooKeeper 参数被替换了
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("group.id", "test");
props.put("enable.auto.commit", "true");
props.put("auto.commit.interval.ms", "1000");
props.put("session.timeout.ms", "30000");
props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserial
props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeseri
// 相比old consumer 而言，这里创建消费者更加简单
KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
```

```
consumer.subscribe(Arrays.asList("foo", "bar"));
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records)
       System.out.printf("offset = %d, key = %s, value = %s", record.offset(), reco
```

Old Consumer （High Level）：



```
//  old consumer 需要 ZooKeeper
Properties props = new Properties();
props.put("zookeeper.connect", "localhost:2181");
props.put("group.id", "test");
```

```
props.put("auto.commit.enable", "true");
props.put("auto.commit.interval.ms", "1000");
props.put("auto.offset.reset", "smallest");
ConsumerConfig config = new ConsumerConfig(props);
// 需要创建connector
ConsumerConnector connector = Consumer.createJavaConsumerConnector(config);
// 创建message stream
Map<String, Integer> topicCountMap = new HashMap<String, Integer>();
topicCountMap.put("foo", 1);
Map<String, List<KafkaStream<byte[], byte[]>>> streams =
 connector.createMessageStreams(topicCountMap);
// 获取数据
KafkaStream<byte[], byte[]> stream = streams.get("foo").get(0);
ConsumerIterator<byte[], byte[]> iterator = stream.iterator();
MessageAndMetadata<byte[], byte[]> msg = null;
while (iterator.hasNext()) {
     msg = iterator.next();
     System.out.println(//
            " group " + props.get("group.id") + //
            ", partition " + msg.partition() + ", " + //
             new String(msg.message()));
}
```

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

兼容性说明

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

CKafka 版本选择建议

最近更新时间：2024-01-09 15:02:47

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

概述

社区版 Kafka 目前共演进了0.7.x到2.8.x大概20个版本，从消息队列的角度可分为三个阶段：0.x、1.x、2.x。 目前腾讯云针对这三个社区发展阶段提供了四个对应版本：0.10、1.1、2.4、2.8，基本覆盖了用户使用的主流 Kafka 版本。
其中1.x和2.x这两个大版本主要是对 Kafka Streams 的优化和改进，在消息引擎方面并未引入太多的重大功能特性（2.x在事务特性方面有较大改进）。Kafka Streams 在2.x版本有较大改进，如果您是这些特性的用户，请至少选择2.x的版本。

兼容性说明

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

CKafka 版本	可兼容社区版本	兼容性
0.10.2	≤ 0.10.x	100%
1.1.1	≤ 1.1.x	100%
2.4.1	≤ 2.4.x	100%
2.8.1	≤ 2.8.x	100%

关于 CKafka2.4.1 版本说明

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

CKafka 版本选择建议

如果是自建 Kafka 上云，建议选择对应的大版本即可。例如：自建 Kafka 是1.1.0版本，则选择 CKafka 的1.1版本。当在云上找不到对应的版本时，建议向上选择版本。例如：自建是1.0.0版本，则建议使用1.1.1版本；自建是0.11.x版本，则建议使用版本1.1.1（因为 Broker 的每个版本是向下兼容的）。

CKafka 数据可靠性说明

最近更新时间：2024-01-09 15:02:48

本文将分别通过生产端、服务端（CKafka）和消费端介绍影响消息队列 CKafka 版可靠性的因素，并提供对应的解决方法。

生产端数据丢失如何处理？

数据丢失原因

生产者将数据发送到消息队列 CKafka 版时，数据可能因为网络抖动而丢失，此时消息队列 CKafka 版未收到该数据。可能情况：
网络负载高或者磁盘繁忙时，生产者又没有重试机制。
磁盘超过购买规格的限制，例如实例磁盘规格为9000GB，在磁盘写满后未及时扩容，会导致数据无法写入到消息队列 CKafka 版。
突发或持续增长峰值流量超过购买规格的限制，例如实例峰值吞吐规格为100MB/s，在长时间峰值吞吐超过限制后未及时扩容，会导致数据写入消息队列 CKafka 版变慢，生产者有排队超时机制时，导致数据无法写入到消息队列 CKafka 版。

解决方法

生产者对自己重要的数据，开启失败重试机制。
针对磁盘使用，在配置实例时设置好监控和 告警策略 ，可以做到事先预防。
遇到磁盘写满时，可以在控制台及时升配（消息队列 CKafka 版非独占实例间升配为平滑升配不停机且也可以单独升配磁盘）或者通过修改消息保留时间降低磁盘存储。
为了尽可能减少生产端消息丢失，您可以通过 buffer.memory 和 batch.size（以字节为单位）调优缓冲区的大小。缓冲区并非越大越好，如果由于某种原因生产者宕机了，那么缓冲区存在的数据越多，需要回收的垃圾越多，恢复就会越慢。应该**时刻注意生产者的生产消息数情况、平均消息大小**等（消息队列 CKafka 版监控中有丰富的监控指标）。
配置生产端 ACK：
当 producer 向 leader 发送数据时，可以通过 request.required.acks 参数以及 min.insync.replicas 设置数据可靠性的级别。
当 acks = 1时（默认值），生产者在 ISR 中的 leader 已成功收到数据可以继续发送下一条数据。如果 leader 宕机，由于数据可能还未来得及同步给其 follower，则会丢失数据。
当 acks = 0时，生产者不等待来自 broker 的确认就发送下一条消息。这种情况下数据传输效率最高，但数据可靠性最低。
注意：
当生产端配置 acks = 0 时，如果当前实例被限流，为了保护服务端能正常提供服务，服务端会主动关闭与客户端的连接。

当 acks = -1或者 all 时，生产者需要等待 ISR 中的所有 follower 都确认接收到消息后才能发送下一条消息，可靠性最高。
即使按照上述配置 ACK，也不能保证数据不丢，例如，当 ISR 中只有 leader 时（ISR 中的成员由于某些情况会增加也会减少，最少时只剩一个 leader），此时会变成 acks = 1的情况。所以需要同时在配合 min.insync.replicas 参数（此参数可以在消息队列 CKafka 版控制台 Topic 配置开启高级配置中进行配置），min.insync.replicas 表示在 ISR 中最小副本的个数，默认值是1，当且仅当 acks = -1或者 all 时生效。

建议配置的参数值

此参数值仅供参考，实际数值需要依业务实际情况而定。
重试机制：`message.send.max.retries=3;retry.backoff.ms=10000;`
高可靠的保证：`request.required.acks=-1;min.insync.replicas=2;`
高性能的保证：`request.required.acks=0;`
可靠性+性能：`request.required.acks=1;`

服务端（CKafka）数据丢失如何处理？

数据丢失原因

partition 的 leader 在未完成副本数 followers 的备份时就宕机，即使选举出了新的 leader 但是数据因为未来得及备份就丢失。
开源 Kafka 的落盘机制为异步落盘，也就是数据是先存在 PageCache 中的，当还没有正式落盘时，broker 出现断开连接或者重启或者故障时，PageCache 上的数据由于没有来及落磁盘进而丢失。
磁盘故障导致已经落盘的数据丢失。

解决方法

开源 Kafka 是多副本的，官方推荐通过副本来保证数据的完整性，此时如果是多副本，同时出现多副本多 broker 同时挂掉才会丢数据，比单副本数据的可靠性高很多，所以消息队列 CKafka 版强制 Topic 是双副本，可配置3副本。
消息队列 CKafka 版服务配置了更合理的参数 log.flush.interval.messages 和 log.flush.interval.ms，对数据进行刷盘。
消息队列 CKafka 版对磁盘做了特殊处理，保证部分磁盘损坏时也不会影响数据的可靠性。

建议配置的参数值

非同步状态的副本可以选举为 leader：`unclean.leader.election.enable=false // 关闭`

消费端数据丢失如何处理？

数据丢失原因

还未真正消费到数据就提交 commit 了 offset，若过程中消费者挂掉，但 offset 已经刷新，消费者错过了一条数据，需要消费分组重新设置 offset 才能找回数据。
消费速度和生产速度相差太久，而消息保存时间太短，导致消息还未及时消费就被过期删除。

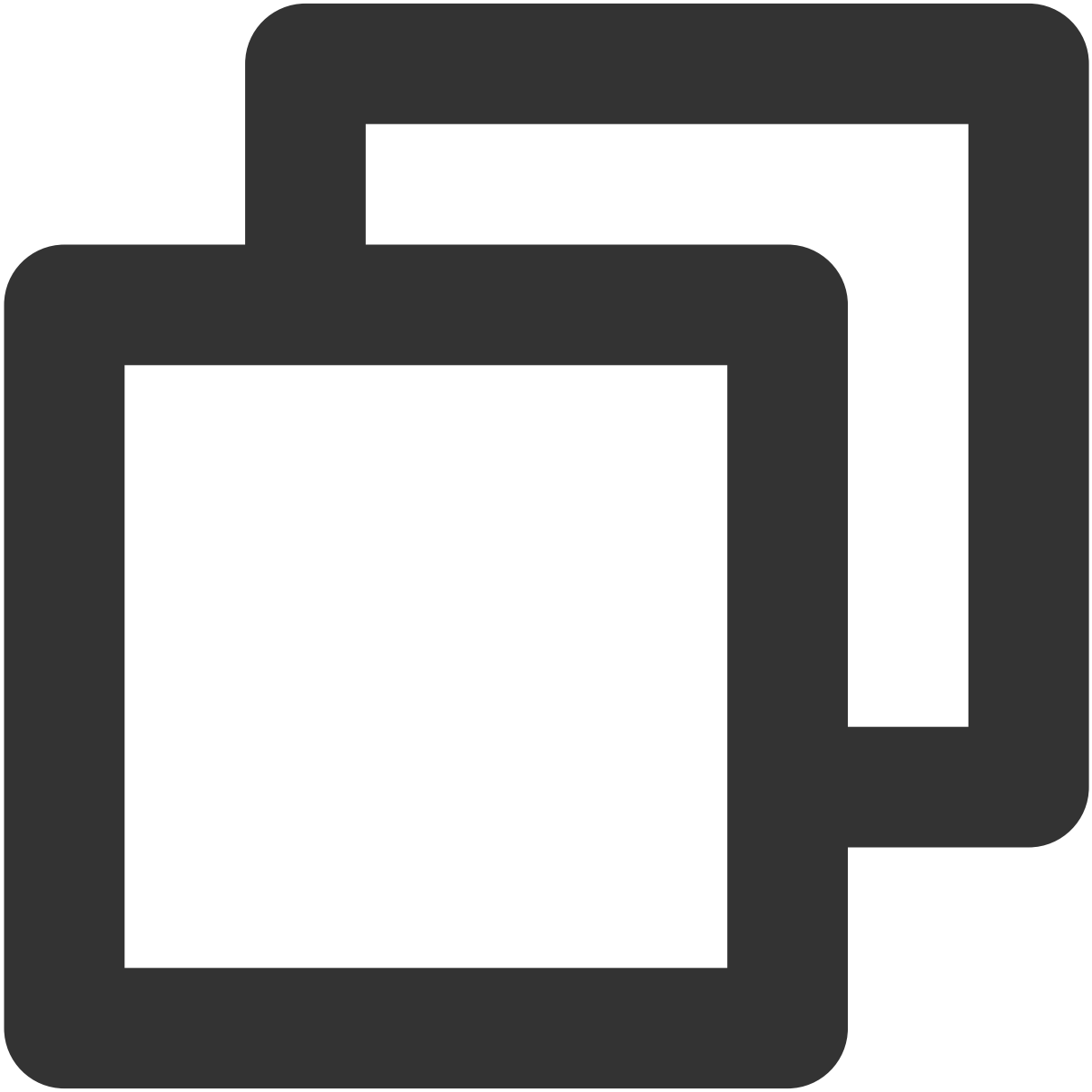
解决方法

合理配置参数 auto.commit.enable，等于 true 时表示自动提交。建议使用定时提交，避免频繁 commit offset。
监控消费者的情况，正确调整数据的保留时间。监控当前消费 offset 以及未消费的消息条数，并配置告警，防止由于消费速度过慢导致消息过期删除。

数据丢失排查方案

在本地打印分区 partition 和偏移量 offset 进行排查

打印信息代码如下：



```
Future<RecordMetadata> future = producer.send(new ProducerRecord<>(topic, messageKe
```

```
RecordMetadata recordMetadata = future.get();
log.info("partition: {}", recordMetadata.partition());
log.info("offset: {}", recordMetadata.offset());
```

如果能够打印出 partition 和 offset，则表示当前发送的消息在服务端已经被正确保存。此时可以通过消息查询的工具去查询相关位点的消息即可。

如果打印不出 partition 和 offset，则表示消息没有被服务端保存，客户端需要重试。

连接器
数据库变更订阅
MySQL 订阅消息 canal 格式说明

最近更新时间：2024-01-09 15:02:47

概述

使用 CKafka 连接器订阅 MySQL 的变更操作时，可选多种消息格式，默认采用 debezium 格式，同时提供了兼容其他消息格式的能力。本文介绍兼容**官方自定义格式**的消息格式说明。

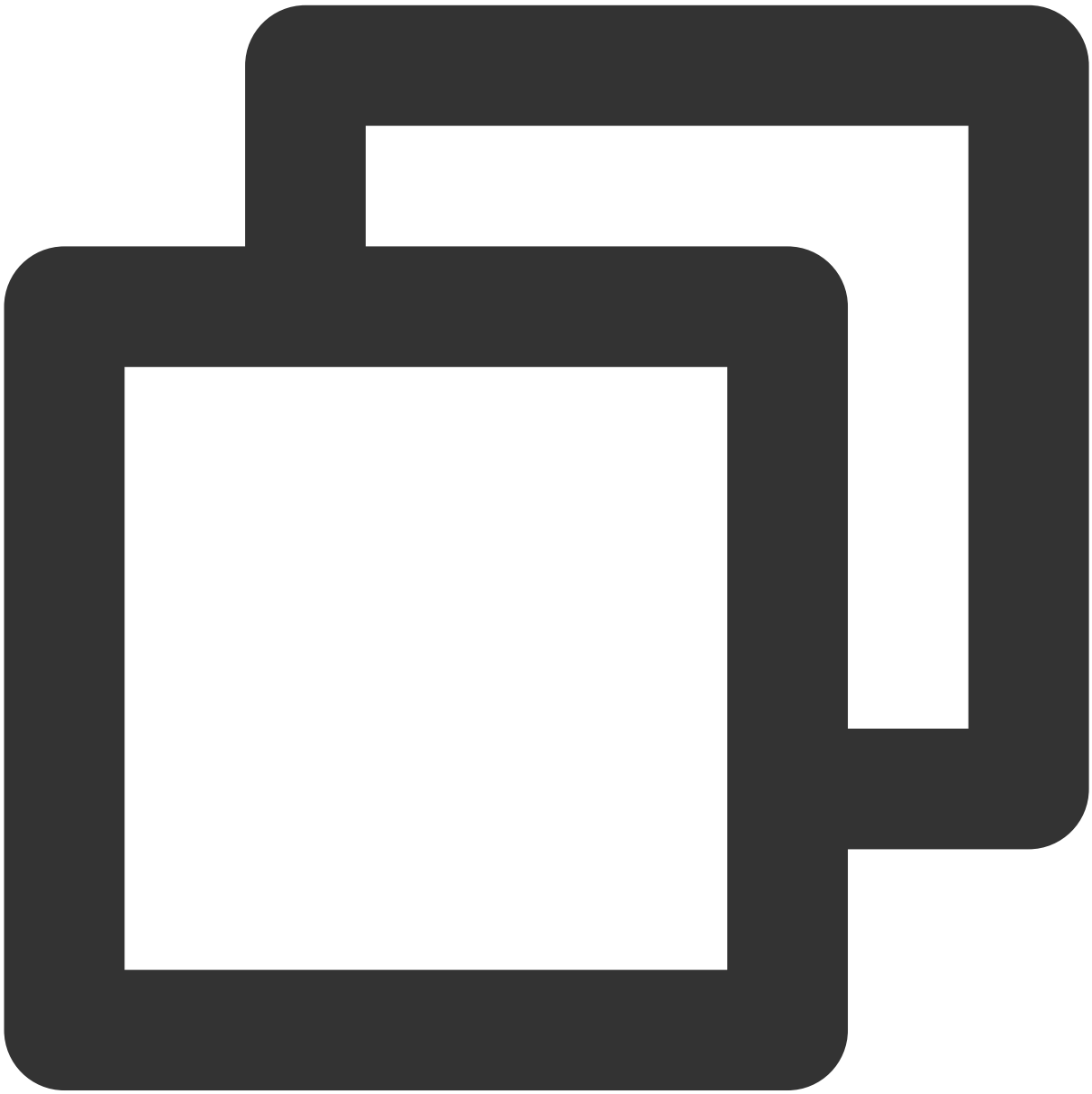
官方格式一说明

官方格式一目前仅支持 DML 消息，DDL 消息格式与 canal 格式一致。

字段名称	字段说明
BINLOG_NAME	binlog 日志文件名称
BINLOG_POS	binlog 日志的 pos 位置
DATABASE	数据库名称
EVENT_SERVER_ID	暂时默认为 null
GLOBAL_ID	如果开启 GTID，则为 GTID 信息
GROUP_ID	暂时默认为 null
NEW_VALUES	type=U，则为更新后的行信息，格式为 json type=D，则为 null type=I，则为新插入的行信息，格式为 json
OLD_VALUES	type=U，则为更新前的行信息，格式为 json type=D，则为删除的行信息，格式为 json type =I，则为 null
TABLE	表名
TIME	日志生成时间
TYPE	日志类型。U：update，D：delete，I：insert

DDL 格式

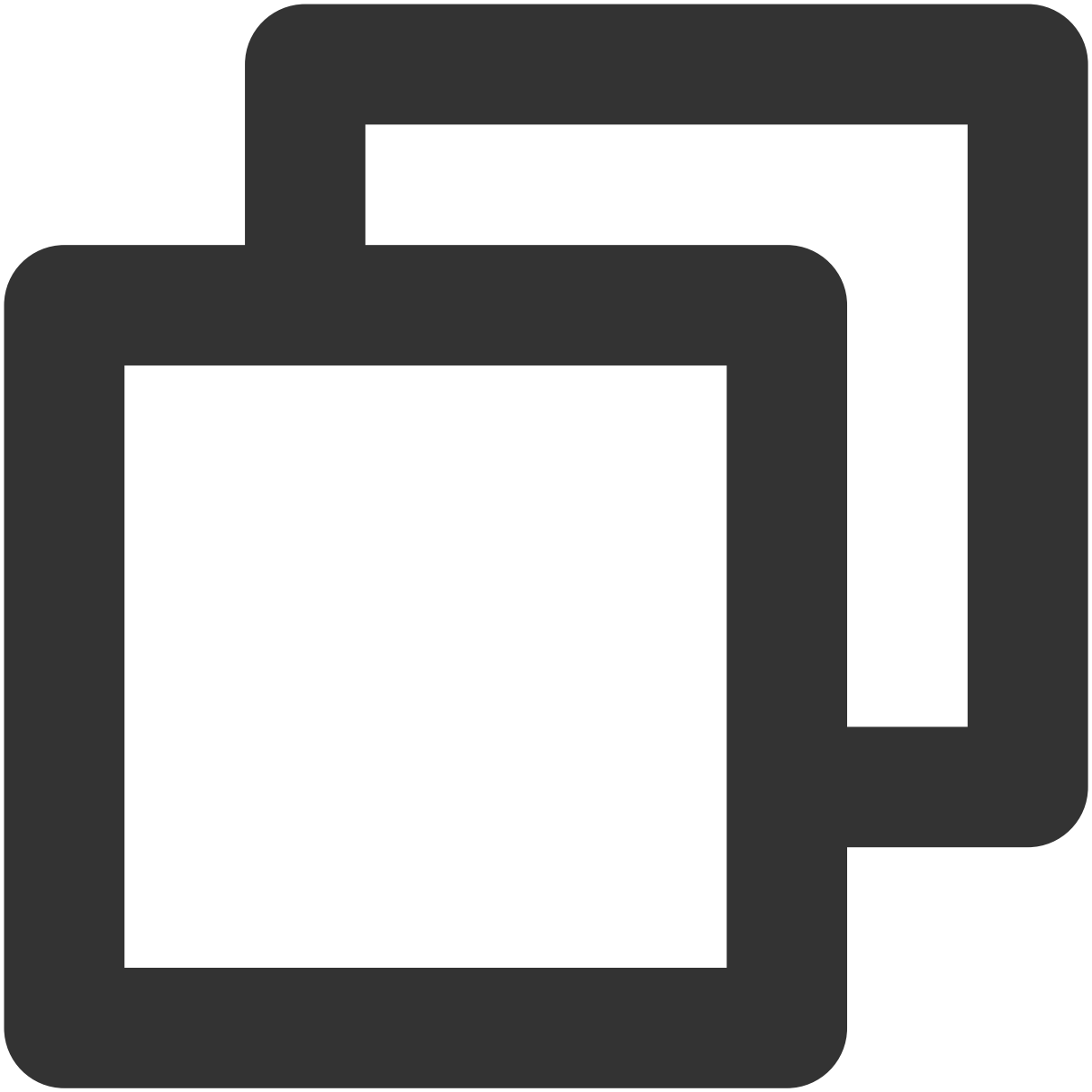
create database



```
{
    "data": null,
    "database": "dip_test",
    "es": 1655812326,
```

```
    "id": 0,
    "isDdl": true,
    "mysqlType": null,
    "old": null,
    "pkNames": null,
    "sql": "CREATE DATABASE `dip_test` CHARSET utf8mb4 COLLATE utf8mb4_0900_ai_ci",
    "sqlType": null,
    "table": "",
    "ts": 1655812326,
    "type": "QUERY"
}
```

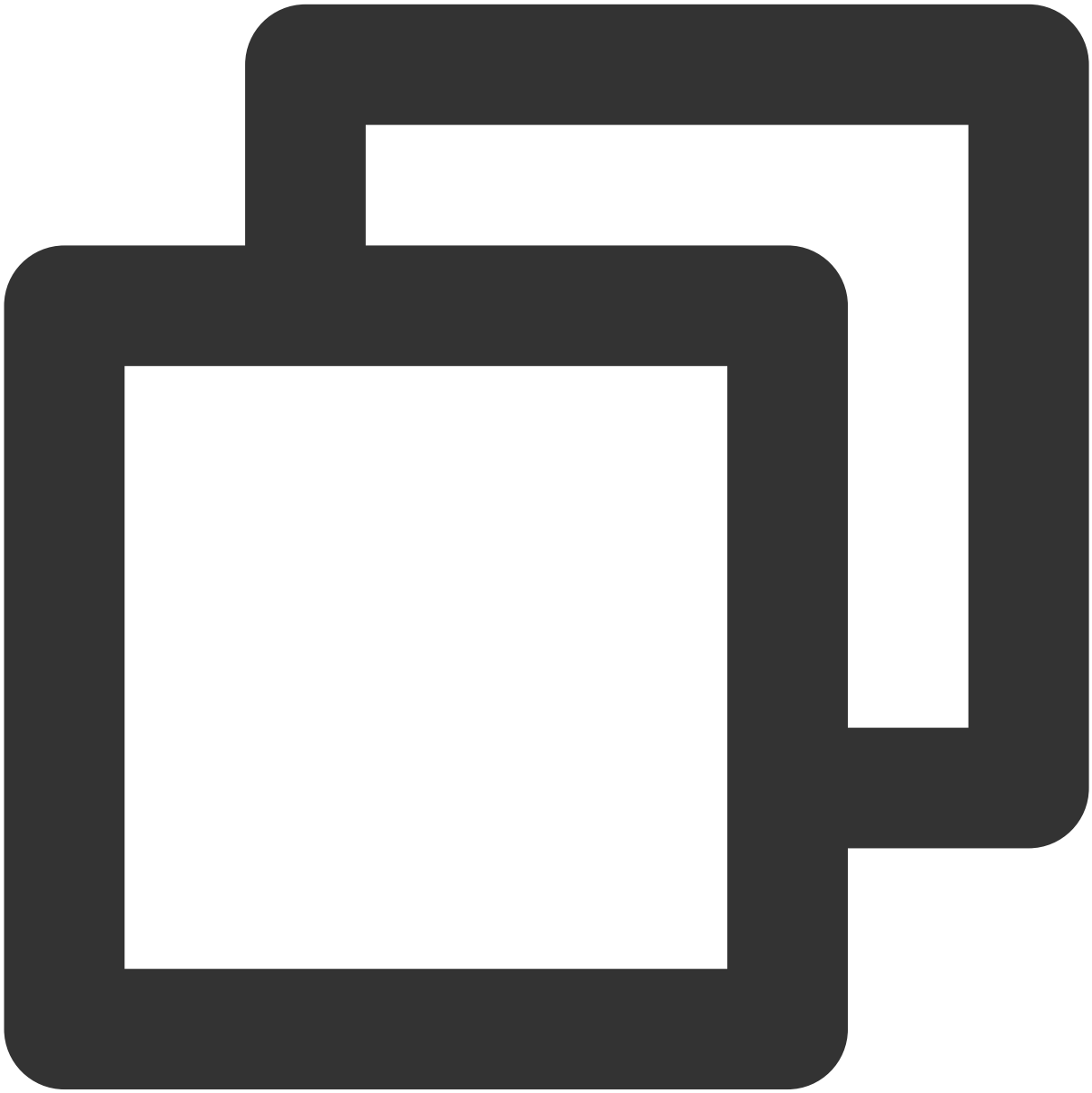

drop database




```
{
    "data": null,
    "database": "dip_test",
    "es": 1655812326,
    "id": 0,
    "isDdl": true,
    "mysqlType": null,
    "old": null,
    "pkNames": null,
    "sql": "DROP DATABASE IF EXISTS `dip_test`",
    "sqlType": null,
```

```
    "table": "",
    "ts": 1655812326,
    "type": "QUERY"
}
```

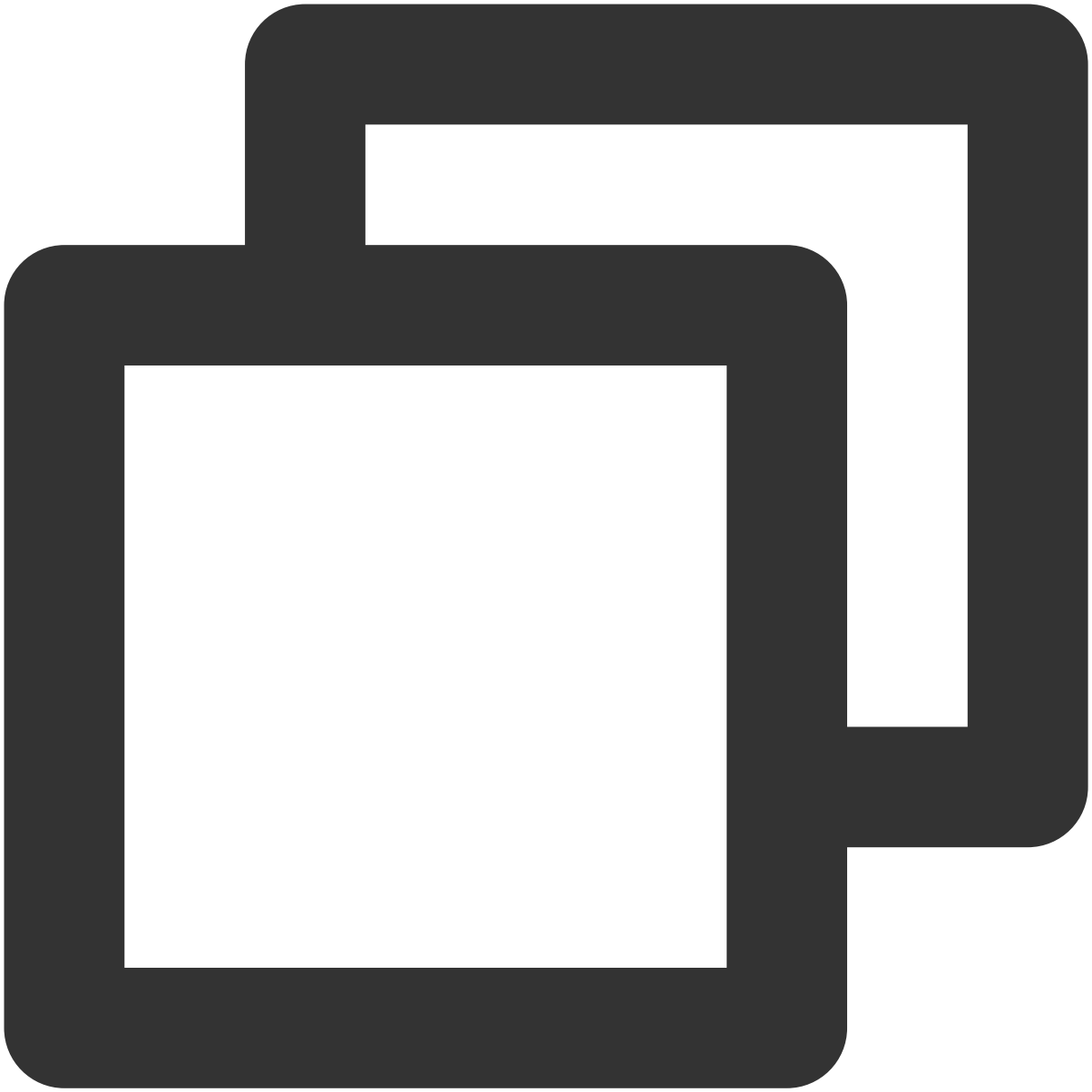
create table



```
{
    "data": null,
    "database": "dip_test",
```

```
    "es": 1655812326,
    "id": 0,
    "isDdl": true,
    "mysqlType": null,
    "old": null,
    "pkNames": null,
    "sql": "CREATE TABLE `customers` (
  `id` int NOT NULL AUTO_INCREMENT,
  `first_name` varchar(255) NOT NULL,
  `last_name` varchar(255) NOT NULL,
  `email` varchar(255) NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `email` (`email`),
  KEY `ix_id` (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=1041 DEFAULT CHARSET=utf8",
    "sqlType": null,
    "table": "customers",
    "ts": 1655812326,
    "type": "CREATE"
}
```

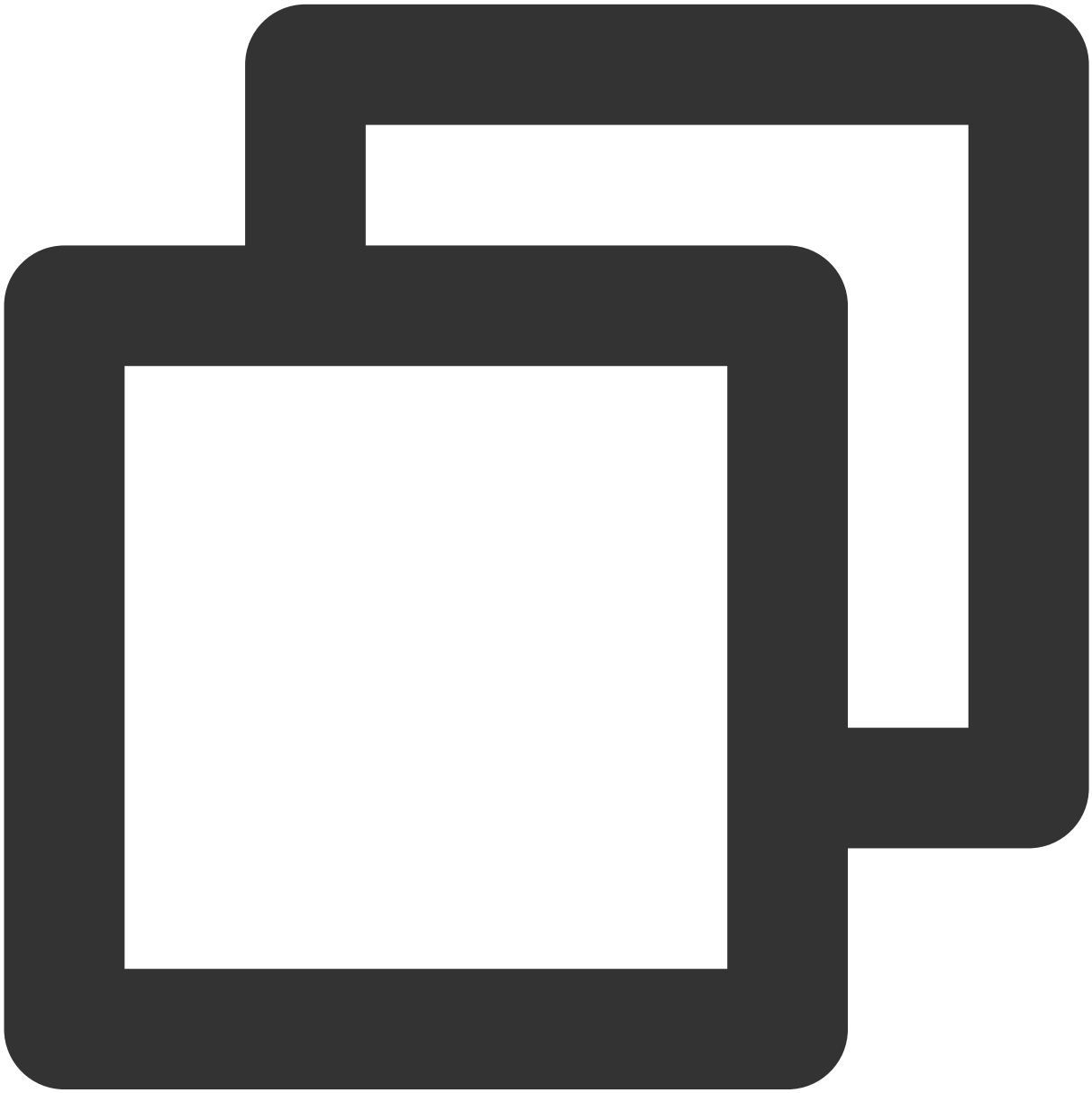
alter table




```
{
    "data": null,
    "database": "test",
    "es": 1655782153,
    "id": 0,
    "isDdl": true,
    "mysqlType": null,
    "old": null,
    "pkNames": null,
    "sql": "ALTER TABLE `user` ADD COLUMN `createtime` datetime NULL DEFAULT CURREN
    "sqlType": null,
```

```
    "table": "user",
    "ts": 1655782153,
    "type": "ALTER"
}
```

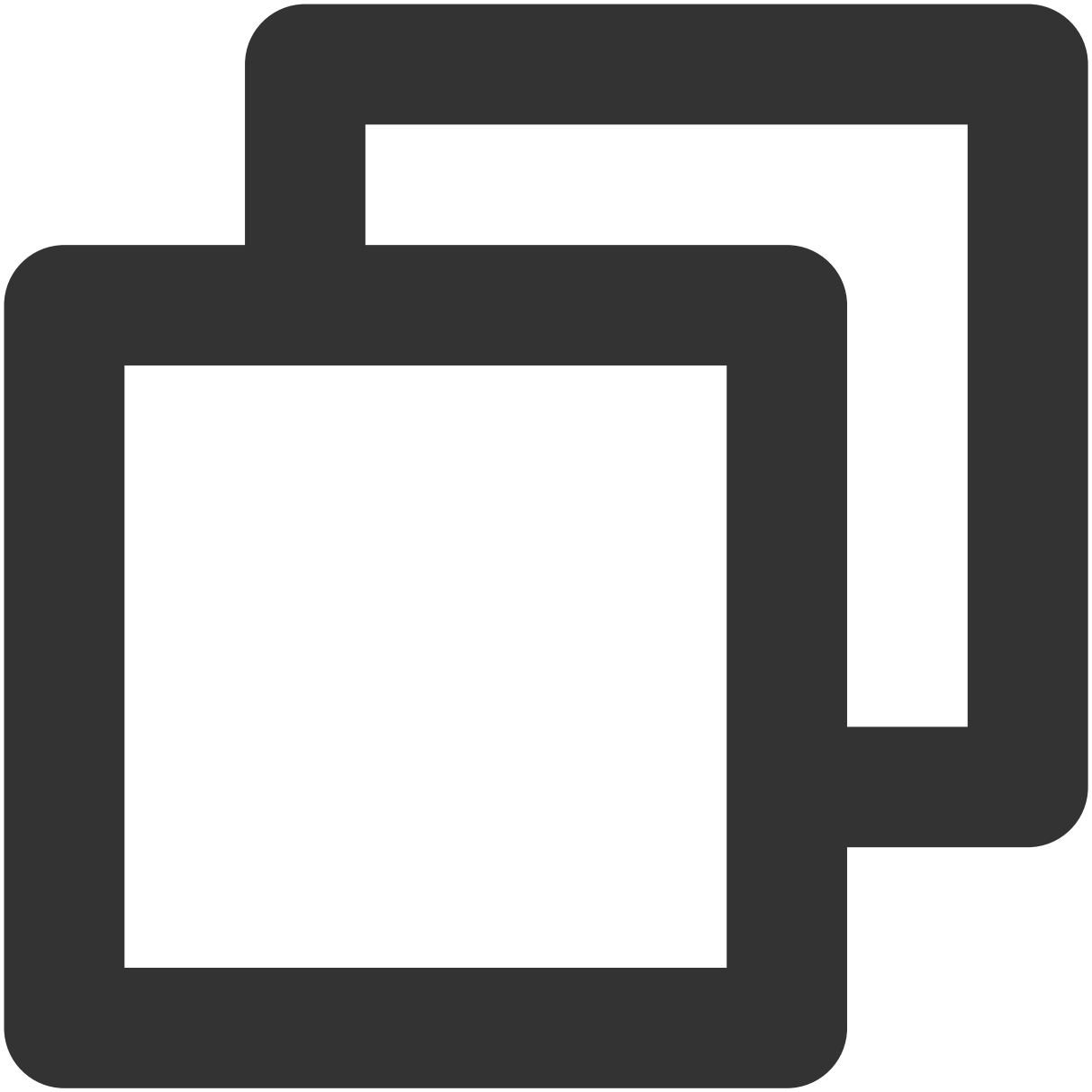
drop table



```
{
    "data": null,
    "database": "dip_test",
```

```
    "es": 1655812326,
    "id": 0,
    "isDdl": true,
    "mysqlType": null,
    "old": null,
    "pkNames": null,
    "sql": "DROP TABLE IF EXISTS `dip_test`.`customers`",
    "sqlType": null,
    "table": "customers",
    "ts": 1655812326,
    "type": "ERASE"
}
```

rename table

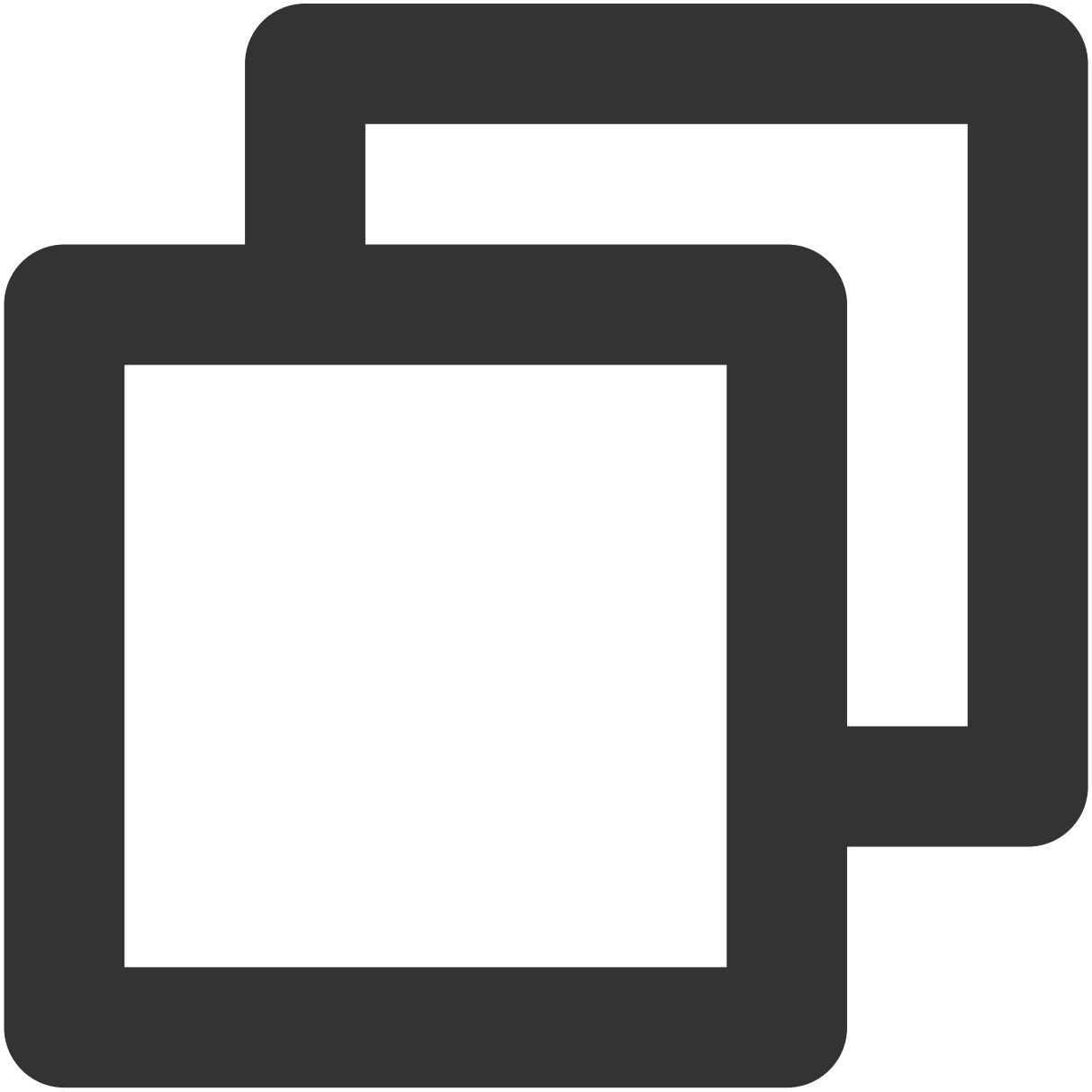


```
{
    "data": null,
    "database": "testDB",
    "es": 1656300979748,
    "id": 0,
    "isDdl": true,
    "mysqlType": null,
    "old": null,
    "pkNames": null,
    "sql": "rename table test to t_test",
    "sqlType": null,
```

```
    "table": "t_test",
    "ts": 1656300979748,
    "type": "RENAME"
}
```

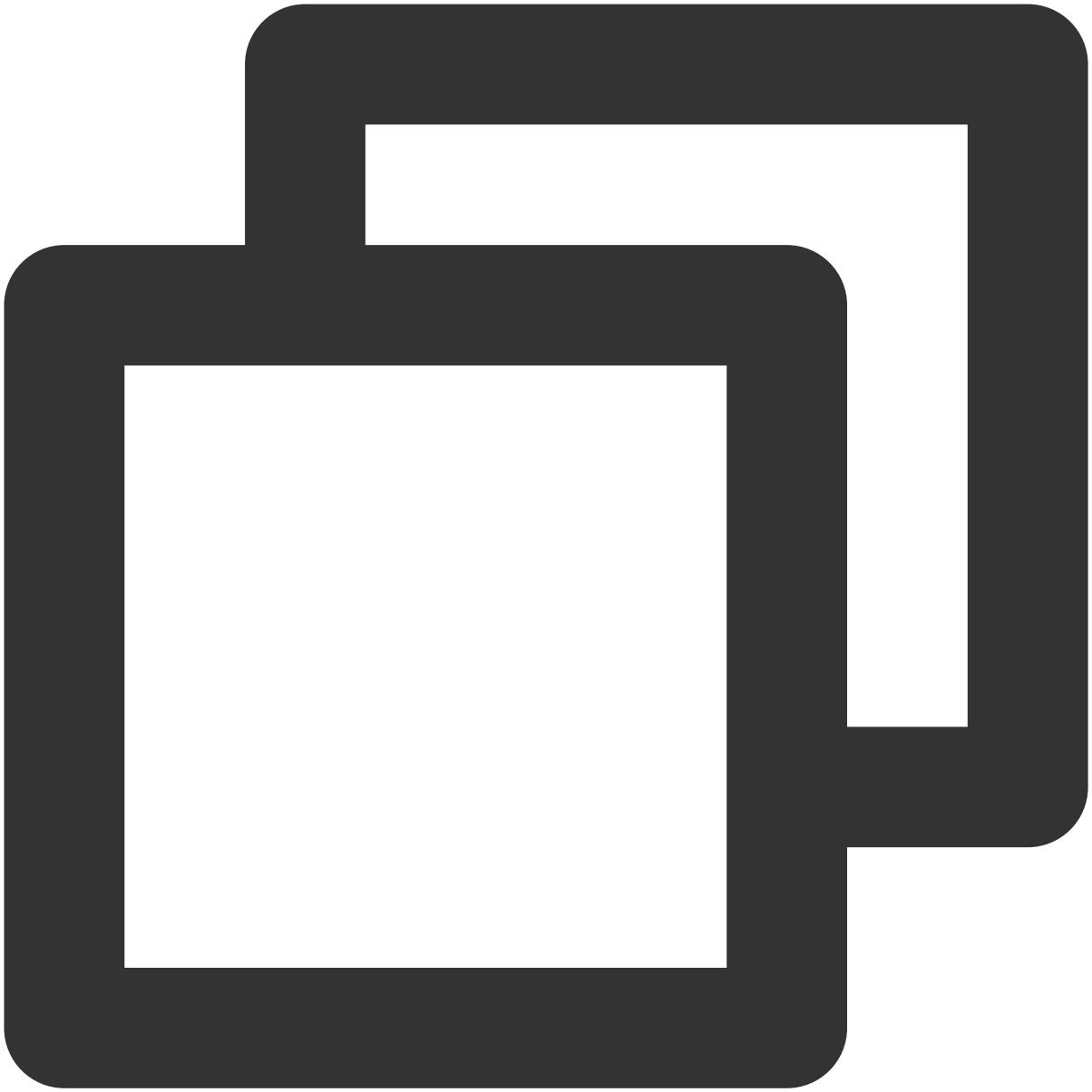
DML 格式

insert



```
{
    "BINLOG_NAME": "mysql-bin.000003",
    "BINLOG_POS": 154,
    "DATABASE": "inventory",
    "EVENT_SERVER_ID": null,
    "GLOBAL_ID": null,
    "GROUP_ID": null,
    "NEW_VALUES": {
        "last_name": "Kretchmar",
        "id": "1004",
        "first_name": "Anne",
        "email": "annek@noanswer.org"
    },
    "OLD_VALUES": null,
    "TABLE": "customers",
    "TIME": "19700101080000",
    "TYPE": "I"
}
```

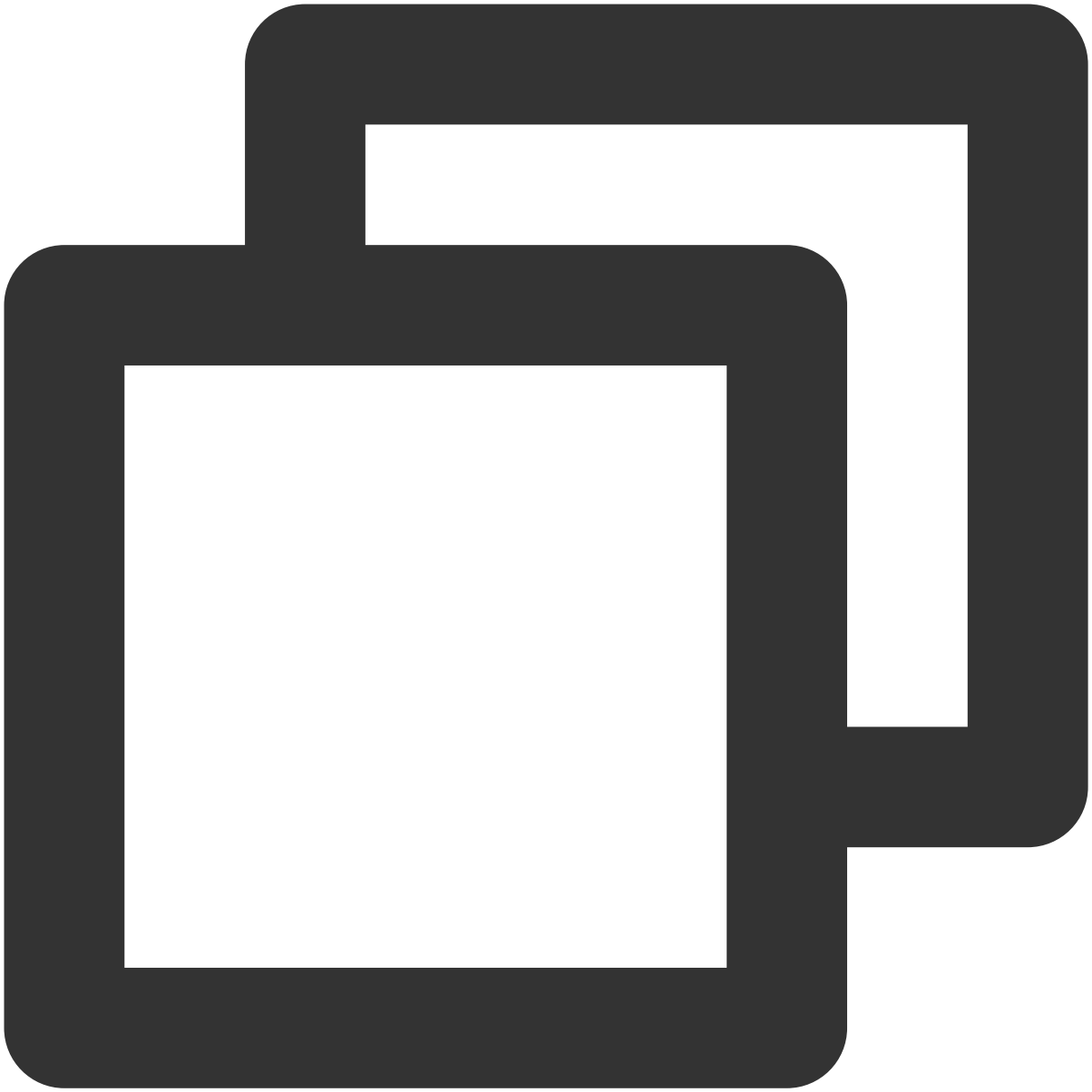
update



```
{
    "BINLOG_NAME": "mysql-bin.000003",
    "BINLOG_POS": 484,
    "DATABASE": "inventory",
    "EVENT_SERVER_ID": null,
    "GLOBAL_ID": null,
    "GROUP_ID": null,
    "NEW_VALUES": {
        "last_name": "Kretchmar",
        "id": "1004",
        "first_name": "Anne Marie",
```

```
        "email": "annek@noanswer.org"
    },
    "OLD_VALUES": {
        "last_name": "Kretchmar",
        "id": "1004",
        "first_name": "Anne",
        "email": "annek@noanswer.org"
    },
    "TABLE": "customers",
    "TIME": "20160611015029",
    "TYPE": "U"
}
```

delete




```
{
    "BINLOG_NAME": "mysql-bin.000003",
    "BINLOG_POS": 805,
    "DATABASE": "inventory",
    "EVENT_SERVER_ID": null,
    "GLOBAL_ID": null,
    "GROUP_ID": null,
    "NEW_VALUES": null,
    "OLD_VALUES": {
        "last_name": "Kretchmar",
        "id": "1004",
```

```
        "first_name": "Anne Marie",
        "email": "annek@noanswer.org"
    },
    "TABLE": "customers",
    "TIME": "20160611020502",
    "TYPE": "D"
}
```