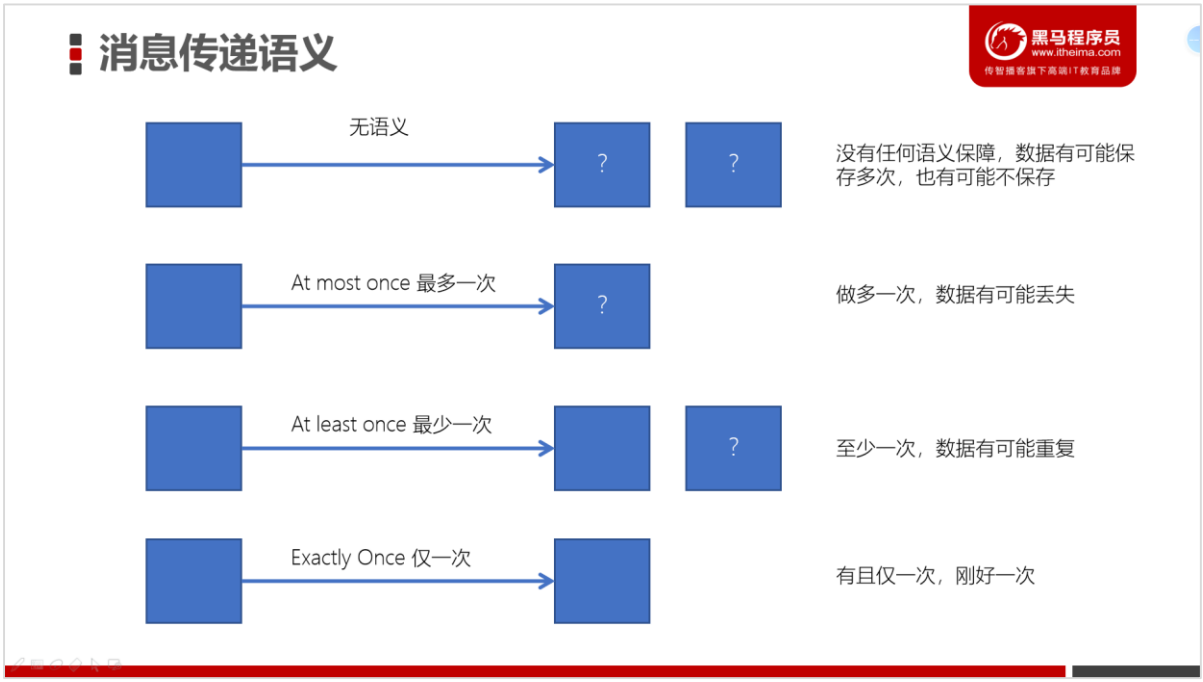
第二章 Kafka高级

1. 消息传递语义

1.1 三种消息的传递语义



1.1.1 At most once语义

At most once表示消息可能会丢失，但绝不会重复传输。

1.1.2 At least once语义

At least once表示消息绝不会丢失，但可能会重复传输。

1.1.3 Exactly Once语义

每条消息肯定会被传输一次且仅传输一次。

1.2 Exactly Once语义保障

1.2.1 高级API与低级API

1.2.1.1 高级API

```
/**
 * 消费者程序：从test主题中消费数据
 */
public class _2ConsumerTest {
    public static void main(String[] args) {
        // 1. 创建Kafka消费者配置
        Properties props = new Properties();
        props.setProperty("bootstrap.servers", "192.168.88.100:9092");
        props.setProperty("group.id", "test");
        props.setProperty("enable.auto.commit", "true");
        props.setProperty("auto.commit.interval.ms", "1000");
        props.setProperty("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
        props.setProperty("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");

        // 2. 创建Kafka消费者
        KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);

        // 3. 订阅要消费的主题
        consumer.subscribe(Arrays.asList("test"));

        // 4. 使用一个while循环，不断从Kafka的topic中拉取消息
```

```
        while (true) {
            // 定义100毫秒超时
            ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
            for (ConsumerRecord<String, String> record : records)
                System.out.printf("offset = %d, key = %s, value = %s%n", record.offset(), record.key(), rec
ord.value());
        }
    }
}
```

- 上面是之前编写的代码，消费Kafka的消息很容易实现，写起来比较简单
- 不需要执行去管理offset,通过ZK管理；也不需要管理分区、副本，由Kafka统一管理
- 消费者会自动根据上一次在ZK中保存的offset去接着获取数据
- 在ZK中，不同的消费者组(group)同一个topic记录不同的offset，这样不同程序读取同一个topic，不会受offset的影响

高级API的缺点

- 不同执行控制offset，例如：想从指定的位置读取
- 不同细化控制分区、副本、ZK等

1.2.1.2 低级API

通过使用低级API，我们可以自己来控制offset，想从哪儿读，就可以从哪儿读。而且，可以自己控制连接分区，对分区自定义负载均衡。而且，之前offset是自动保存在ZK中，使用低级API，我们可以将offset不一定要使用ZK存储，我们可以自己来存储offset。例如：存储在文件、MySQL、或者内存中。但是低级API，比较复杂，需要执行控制offset，连接到哪个分区，并找到分区的leader。

1.3 手动分配消费分区

之前的代码，我们让Kafka根据消费组中的消费者动态地为topic分配要消费的分区。但在某些时候，我们需要对要消费的分区，例如：

- 如果某个程序将某个指定分区的数据保存到外部存储中，例如：Redis、MySQL，那么保存数据

的时候，只需要消费该指定的分区数据即可

- 如果某个程序是高可用的，在程序出现故障时将自动重启(例如：后面我们将学习的Flink、Spark程序)。这种情况下，程序将从指定的分区重新开始消费数据。

如何进行手动消费分区中的数据呢？

1. 不再使用之前的 subscribe 方法订阅主题，而使用 「assign」方法指定想要消费的消息

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

2. 一旦指定了分区，就可以就像前面的示例一样，在循环中调用「poll」方法消费消息

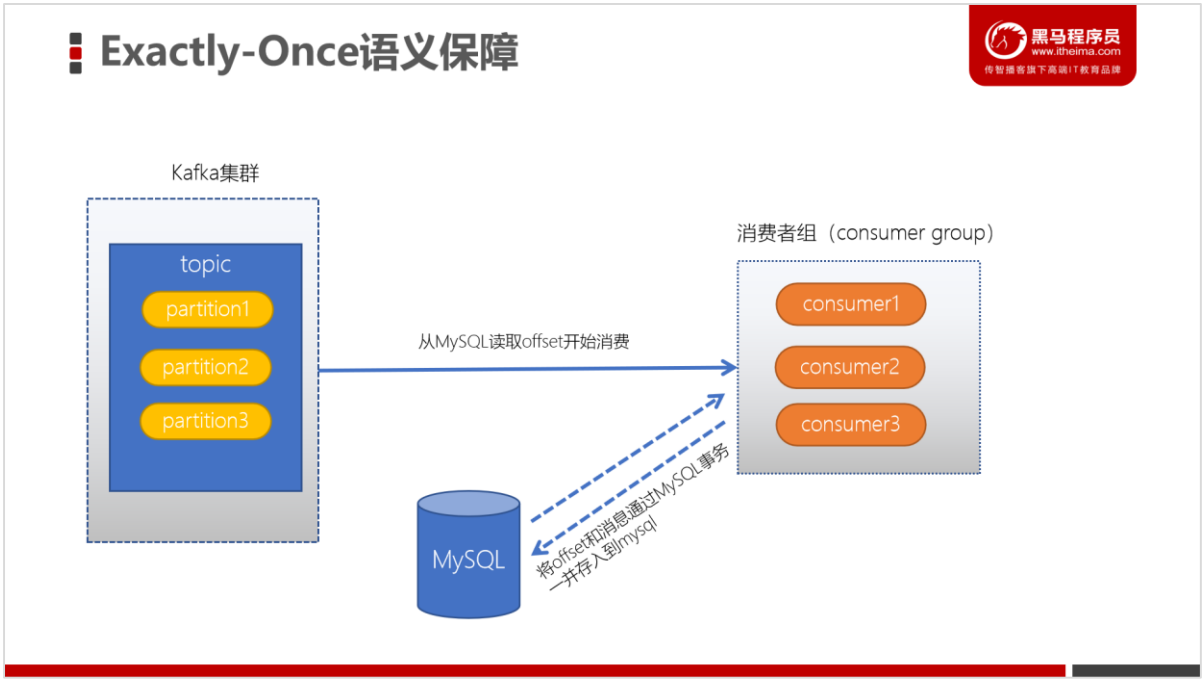
注意

1. 当手动管理消费分区时，即使GroupID是一样的，Kafka的组协调器都将不再起作用
2. 如果消费者失败，也将不再自动进行分区重新分配
3. 为了避免提交的offset冲突，每一个消费者都应该有独立的group Id

1.4 Exactly-Once语义保障

消费者应用程序不需要使用Kafka内置的偏移存储（ZooKeeper），它可以在自己选择的存储中存储偏移量，例如：MySQL、Redis等。这种方式可以实现 Exactly-Once（仅一次），并给出比默认的「至少一次」语义更强的「精确一次」语义。

为了实现「Exactly-Once」，可以将分区的offset保存在MySQL数据库中。如果offset存储在关系数据库中，那么在数据库中存储偏移量可以在单个事务中提交结果和偏移量。故要么事务成功，然后根据所消费的内容更新偏移量，要么不会存储结果，也不会更新偏移量。



1.4.1 实现思路

1. 配置「 enable.auto.commit=false 」，即手动提交

2. 从ConsumerRecord中获取到offset，并保存到数据库中

3. 重启启动程序，从MySQL中读取offset，并使用seek方法消费消息

4. 开启MySQL事务，先保存消息到MySQL表中，然后再将分区的offset保存到到指定的表中。当表中不存在该分区的offset时，插入offset记录。但表中存在该分区的offset时，更新offset记录

1.4.1.1 在MySQL中创建用于保存offset的表

```
-- 创建用于保存Kafka消费者、分区id、偏移量的表
create table if not exists test.t_kafka_topic_consumer_offset
(
    `topic`     varchar(255) comment '主题名称',
    `groupId`   varchar(255) comment '消费者组id',
    `partitionId` integer comment '分区编号',
    `offset`    integer comment '消费到的偏移量',
    primary key (`topic`, `groupId`, `partitionId`)
```

```
);
```

1.4.1.2 准备数据库连接

```
    private static Connection connection = null;
    static {
        try {
            // 1. 加载驱动
            Class.forName("com.mysql.jdbc.Driver");
            // 2. 获取数据库连接
            connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/test", "root", "000000");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

1.4.1.3 将offset保存或更新到MySQL

一开始运行消费者程序的时候，保存offset的表格是空的，所以需要执行insert语句，来将offset保存到表中。但后续如果该记录已经有了，则只需要update就可以了。

注意：因为后续要添加事务保障，所以只需将异常抛出即可。

```
    // 保存offset
    private static void saveConsumerOffset(String groupId, String topic, Integer partition, long offset) throws Exception {
        // 先从MySQL查询是否有topic、partition的offset
        // 1. 如果没有，插入
        // 2. 如果有，更新offset
        String queryOffsetSQL = "select count(*) from test.t_kafka_topic_consumer_offset where topic = ? and groupId = ? and partitionId = ?";
        PreparedStatement psForQueryOffset = connection.prepareStatement(queryOffsetSQL);
        psForQueryOffset.setString(1, topic);
```

```
        psForQueryOffset.setString(2, groupId);
        psForQueryOffset.setInt(3, partition);
        ResultSet rsForQueryOffset = psForQueryOffset.executeQuery();

        boolean hasRecord = false;

        while (rsForQueryOffset.next()) {
            int count = rsForQueryOffset.getInt(1);
            if (count > 0) {
                hasRecord = true;
            } else {
                hasRecord = false;
            }
        }

        if (!hasRecord) {
            String sql = "insert into test.t_kafka_topic_consumer_offset(topic, groupId, partitionId, offset)
VALUES (?, ?, ?, ?)";
            PreparedStatement preparedStatement = connection.prepareStatement(sql);
            preparedStatement.setString(1, topic);
            preparedStatement.setString(2, groupId);
            preparedStatement.setInt(3, partition);
            preparedStatement.setLong(4, offset);

            preparedStatement.execute();
        } else {
            String sql = "update test.t_kafka_topic_consumer_offset set offset = ? where  topic = ? and g
roupId = ? and partitionId = ?";
            PreparedStatement preparedStatement = connection.prepareStatement(sql);
            preparedStatement.setLong(1, offset);
            preparedStatement.setString(2, topic);
            preparedStatement.setString(3, groupId);
            preparedStatement.setInt(4, partition);

            preparedStatement.execute();
        }
    }
```

1.4.1.4 从MySQL中读取出offset

```
/**
 * 从MySQL中读取offset
 *
 * @param topic
 * @param partition
 * @return
 */
private static Integer restoreConsumerOffset(String groupId, String topic, Integer partition) {
    try {// 2. 准备SQL语句
        String sql = "select offset from test.t_kafka_topic_consumer_offset where topic = ? and partitionId = ? and groupId = ?";
        PreparedStatement preparedStatement = connection.prepareStatement(sql);
        preparedStatement.setString(1, topic);
        preparedStatement.setInt(2, partition);
        preparedStatement.setString(3, groupId);

        ResultSet resultSet = preparedStatement.executeQuery();
        while (resultSet.next()) {
            int offset = resultSet.getInt("offset");
            return offset;
        }

        return -1;

    } catch (Exception e) {
        e.printStackTrace();
    }
    return -1;
}
```

1.4.1.5 将消息保存到MySQL中

```
// 将数据插入到MySQL中
private static void insertIntoDb(List<ConsumerRecord<String, String>> buffer) throws Exception {
    // 准备SQL语句
```

```
        String sql = "insert into test.t_kafka_message(`topic`, partitionId, offset, `key`, `value`) VALUES (?,?,?,?,?)";

        PreparedStatement preparedStatement = connection.prepareStatement(sql);

        for (ConsumerRecord<String, String> record : buffer) {
            preparedStatement.setString(1, record.topic());
            preparedStatement.setInt(2, record.partition());
            preparedStatement.setLong(3, record.offset());
            preparedStatement.setString(4, record.key());
            preparedStatement.setString(5, record.value());
            // 3. 执行SQL语句
            preparedStatement.execute();

            System.out.println("key:" + record.key() + " value:" + record.value());
        }
    }
```

1.4.1.6 消费并存储消息

```
        // 指定消费者组的id
        String groupId = "test1";
        // 指定要消费的分区
        String topic = "test";

        Properties props = new Properties();
        props.setProperty("bootstrap.servers", "node1.itcast.cn:9092");
        props.setProperty("group.id", groupId);
        // 手动提交offset
        props.setProperty("enable.auto.commit", "false");
        props.setProperty("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
        props.setProperty("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
        KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);

        TopicPartition partition0 = new TopicPartition(topic, 0);
```

```
        // 指定要消费的分区列表
        consumer.assign(Arrays.asList(partition0));

        // 设置缓存多少个消息，然后再批量提交到MySQL
        final int minBatchSize = 5;
        List<ConsumerRecord<String, String>> buffer = new ArrayList<>();

        // 恢复之前的offset
        Integer offset = restoreConsumerOffset(groupId, topic, partition0.partition());

        // 如果从MySQL获取到offset，则从offset + 1处开始消费，否则，从分区的开始处开始消费
        if (offset != -1) {
            consumer.seek(partition0, offset + 1);
        } else {
            consumer.seek(partition0, 0);
        }

        while (true) {
            ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
            for (ConsumerRecord<String, String> record : records) {
                buffer.add(record);
            }
            if (buffer.size() >= minBatchSize) {
                try {
                    // 开启事务
                    connection.setAutoCommit(false);
                    // 保存消息
                    insertIntoDb(buffer);
                    // 保存offset
                    saveConsumerOffset(groupId, topic, partition0.partition(), buffer.get(buffer.size() - 1).off
set());
                    // 提交事务
                    connection.commit();
                } catch (Exception e) {
                    // 回滚事务
                    e.printStackTrace();
                    connection.rollback();
```

```
            }
            consumer.commitSync();
            //清空内存中的缓存
            buffer.clear();
        }
    }
```

2. 监控工具Kafka-eagle介绍

2.1 Kafka-Eagle简介

在开发工作中，当业务前提不复杂时，可以使用Kafka命令来进行一些集群的管理工作。但如果业务变得复杂，例如：我们需要增加group、topic分区，此时，我们再使用命令行就感觉很不方便，此时，如果使用一个可视化的工具帮助我们完成日常的管理工作，将会大大提高对于Kafka集群管理的效率，而且我们使用工具来监控消费者在Kafka中消费情况。

早期，要监控Kafka集群我们可以使用Kafka Monitor以及Kafka Manager，但随着我们对监控的功能要求、性能要求的提高，这些工具已经无法满足。

Kafka Eagle是一款结合了目前大数据Kafka监控工具的特点，重新研发的一块开源免费的Kafka集群优秀的监控工具。它可以非常方便的监控生产环境中的offset、lag变化、partition分布、owner等。

官网地址：https://www.kafka-eagle.org/

2.2 安装Kafka-Eagle

2.2.1 开启ZooKeeper、Kafka JMX端口

2.2.1.1 JMX接口

JMX(Java Management Extensions)是一个为应用程序植入管理功能的框架。JMX是一套标准的代理和服务，实际上，用户可以在任何Java应用程序中使用这些代理和服务实现管理。很多的一些软

件都提供了JMX接口，来实现一些管理、监控功能。

2.2.1.2 开启Kafka JMX

在启动Kafka的脚本前，添加：

```
cd ${KAFKA_HOME}
export JMX_PORT=9988
nohup bin/kafka-server-start.sh config/server.properties &
```

2.2.2 安装Kafka-Eagle

1. 安装JDK，并配置好JAVA_HOME。

2. 将kafka_eagle上传并解压到 /export/server 目录中。

```
tar -zxvf kafka-eagle-bin-1.4.6.tar.gz
```

3. 配置 kafka_eagle 环境变量。

```
vim /etc/profile
export KE_HOME=/data/soft/new/kafka-eagle
export PATH=$PATH:$KE_HOME/bin
```

4. 配置 kafka_eagle。

```
# 设置集群的名字
kafka.eagle.zk.cluster.alias=cluster1
# 设置kafka地址
cluster1.zk.list=node1.itcast.cn:2181,node2.itcast.cn:2181,node3.itcast.cn:2181
# 设置数据库名字
kafka.eagle.url=jdbc:sqlite:/export/server/kafka-eagle-bin-1.4.6/db/ke.db
# 开启度量统计
kafka.eagle.metrics.charts=true
```

5. 配置JAVA_HOME

vim ke.sh

export JAVA_HOME=/export/server/jdk1.8.0_241/

6. 启动 kafka_eagle。

cd ${KE_HOME}/bin

chmod +x ke.sh

./ke.sh start

7. 访问Kafka eagle，默认用户为admin，密码为：123456

http://192.168.88.100:8048/ke

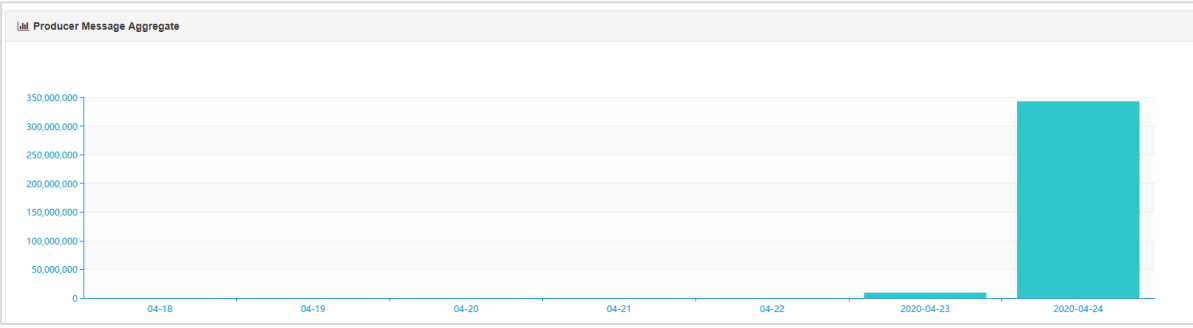
2.3 Kafka度量指标

2.3.1.1 topic list

ID	Topic Name	Partitions	Broker Spread	Broker Skewed	Broker Leader Skewed
1	wordcount	1	33%	0%	0%
2	didi_log	3	100%	0%	33%
3	test	3	100%	0%	33%
4	__transaction_state	50	100%	0%	0%
5	test_10m	3	100%	0%	33%
6	dwd_user	1	33%	0%	0%
7	ods_user	1	33%	0%	0%

Showing 1 to 7 of 7 entries

指标	意义
Brokers Spread	broker使用率
Brokers Skew	分区是否倾斜
Brokers Leader Skew	leader partition是否存在倾斜



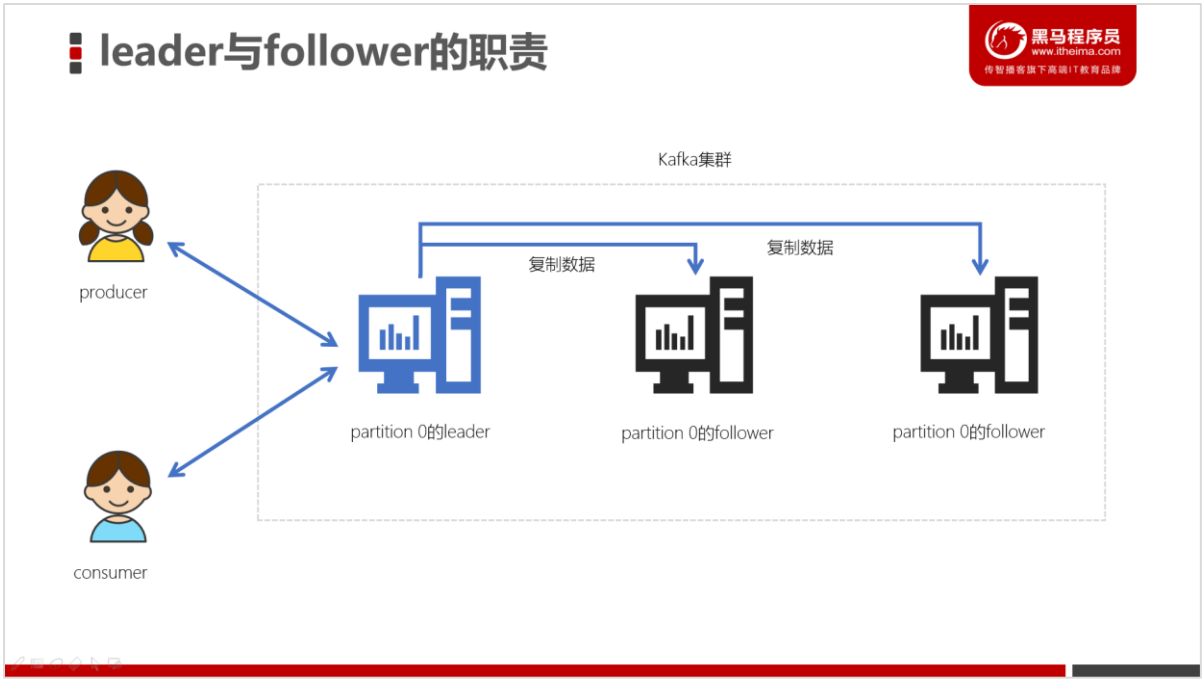
生产者消息总计

3. Kafka原理

3.1 分区的leader与follower

3.1.1 Leader和Follower

在Kafka中，每个topic都可以配置多个分区以及多个副本。同一主题不同分区保存不同的消息，实现了分布式存储。而副本可以保障在某个分区出现故障时，数据仍然是可用的。副本是基于一个分区创建的。



每个分区都有一个leader以及0个或者多个follower，在创建topic时，Kafka会将每个分区的leader均匀地分配在每个broker上。我们正常查看kafka是感觉不到leader、follower的存在的。所有的读写操作都是由leader处理，而所有的follower都复制leader的日志数据文件中，如果leader出现故障时，follower就会被选举为leader。所以，可以这样说：

- **Kafka中的leader负责处理读写操作，而follower只负责副本数据的同步**
- **如果leader出现故障，其他follower会被重新选举为leader**
- **follower像一个consumer一样，拉取leader对应分区的数据，并保存到日志数据文件中。**

3.1.1.1 试一试：查看某个partition的leader

使用Kafka-eagle查看某个topic的partition的leader在哪个服务器中。

Topic Meta Info

Topic	Partition	Log Size	Leader
test	0	4	2
test	1	7	1
test			

分区0的leader在broker2上

Showing 1 to 3 of 3 entries

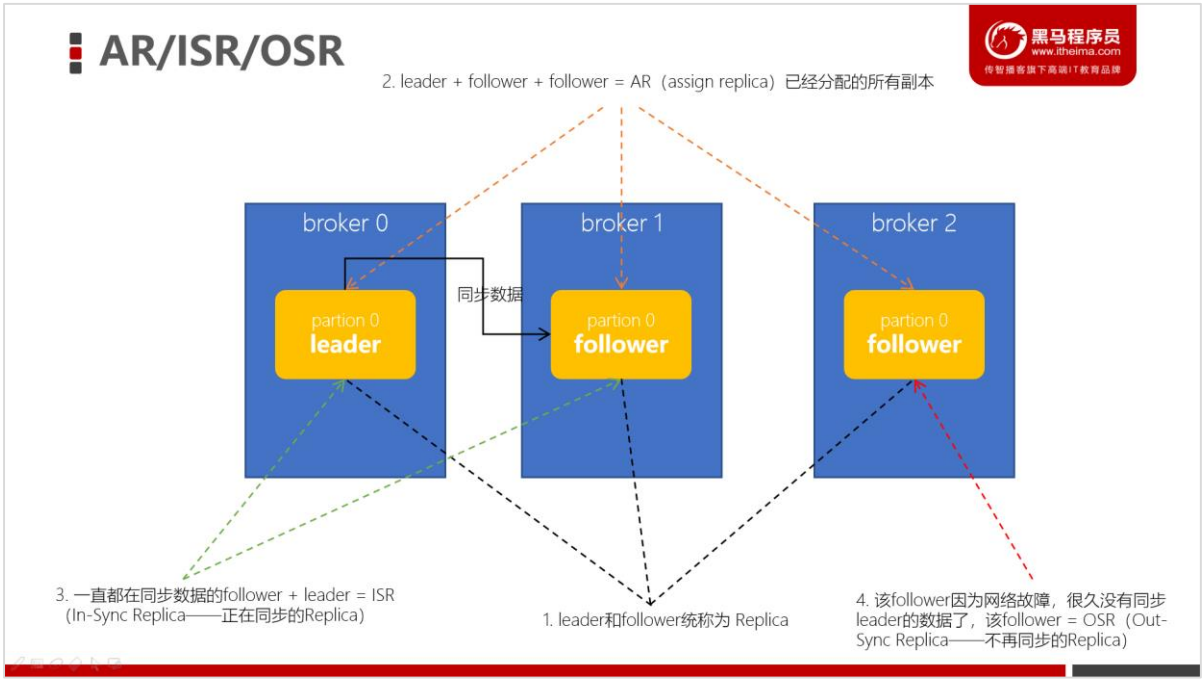
如果leader或follower满足以下两个条件：

1. 可以维护和ZK的连接，ZK可以通过心跳机制检查该leader或follower的连接

2. 如果是follower，必须能够及时同步leader的写操作，延时不能太久

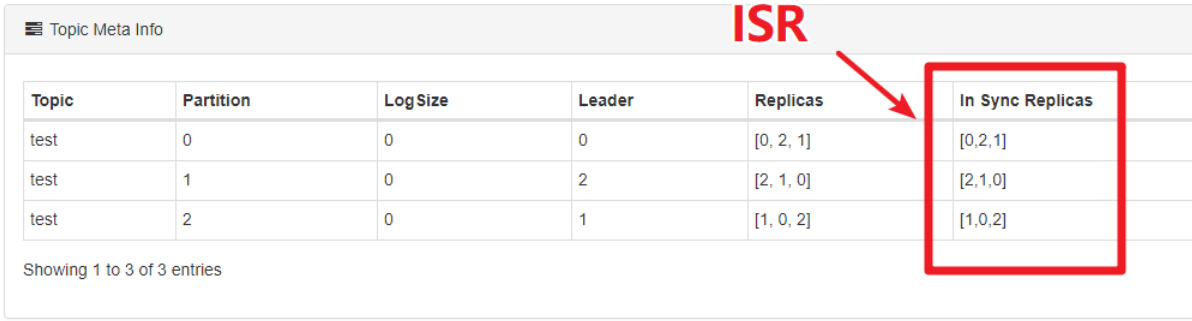
3.1.2 AR、ISR、OSR

在实际环境中，leader、follower都有可能会出现一些故障，在讲解leader选举之前，我们先要明确几个概念。Kafka中，把follower可以按照几个类别来区分不同状态的Replica（副本）。



- 分区的所有副本称为 「AR」（Assigned Replicas——已分配的副本）
- 所有与leader副本保持一定程度同步的副本（包括 leader 副本在内）组成 「ISR」（In-Sync Replicas——在同步中的副本）
- 由于follower副本同步滞后过多的副本（不包括 leader 副本）组成 「OSR」（Out-of-Sync Replied）
- AR = ISR + OSR
- 正常情况下，如果所有的follower副本都应该与leader副本保持一定程度的副本，即AR = ISR，OSR集合为空。

3.1.2.1 试一试：找出分区的ISR



1.
使用Kafka Eagle查看某个Topic的partition的ISR有哪几个节点。

2. 尝试关闭某个partition的Kafka进程，参看topic的ISR情况。

3.2 Leader选举

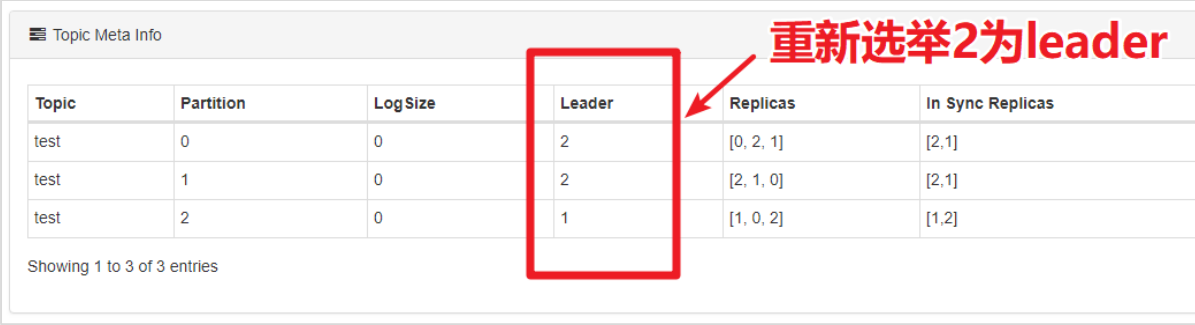
leader对于消息的写入以及读取是非常关键的，此时有两个疑问：

1. Kafka如何确定某个partition是leader、那个partition是follower呢？

2. 因为Kafka的吞吐量是非常高的，一旦某个leader崩溃了，如何快速确定另外一个leader呢？

3.2.1 试一试：leader如果崩溃，Kafka会如何？

使用Kafka Tools找到某个partition的leader，再找到leader所在的broker。在Linux中强制杀掉该Kafka的进程，然后观察leader的情况。



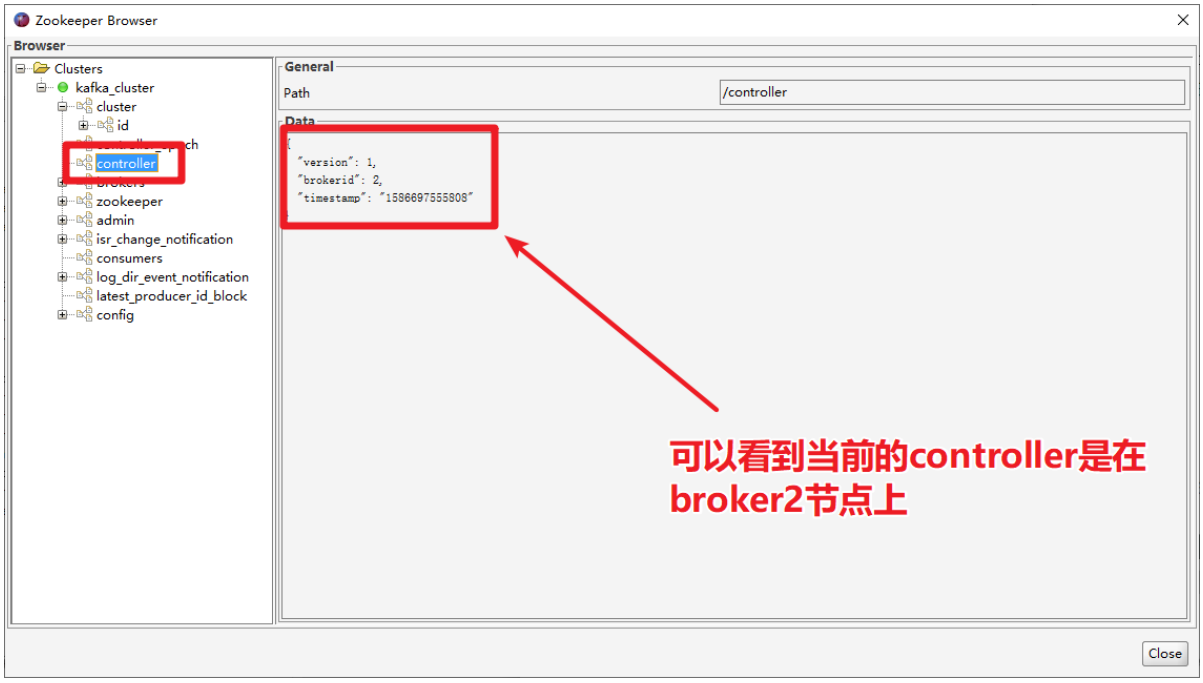
通过观察，我们发现，leader在崩溃后，Kafka又从其他的follower中快速选举出来了leader。

3.2.2 Controller

Kafka启动时，会在所有的broker中选择一个controller，在我们创建topic、或者添加分区、修改副本数量之类的管理任务都是由controller完成的。还有，Kafka分区leader的选举，也是由controller决定的。在Kafka集群启动的时候，每个broker都会尝试去ZooKeeper上注册成为Controller（ZK临时节点），但只有一个竞争成功，其他的broker会注册该节点的监视器，一点该临时节点状态发生变化，就可以进行相应的处理。所以，Controller也是高可用的，一旦某个broker崩溃，其他的broker会重新注册为Controller。

3.2.3 试一试：找到当前Kafka集群的controller

点击Kafka Tools的Tools菜单，找到ZooKeeper Brower...

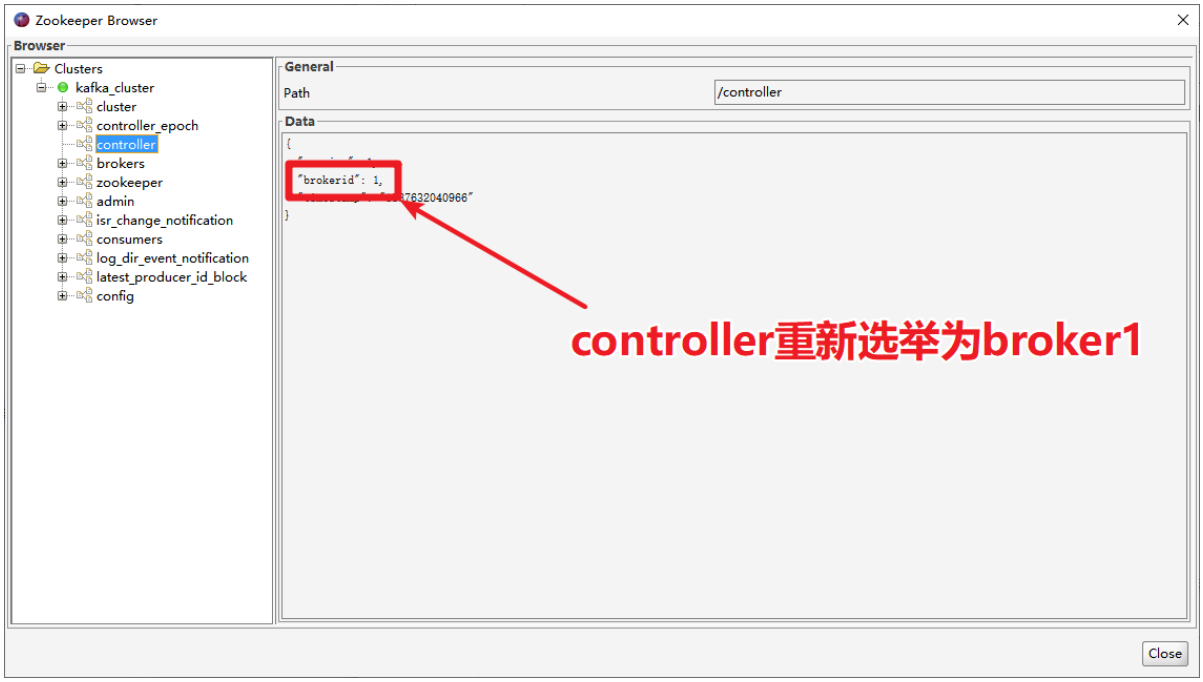


3.2.3.1 Controller选举leader

因为Kafka集群如果业务很多的情况下，会有很多的partition，假设某个broker宕机，就会出现很多的partiton都需要重新选举leader，此时，如果使用zookeeper选举leader，会给zookeeper代码巨大的压力。所以，kafka中leader的选举不能使用ZK来实现。

3.2.3.2 试一试：杀掉controller所在的broker进程

通过kafka tools找到controller所在的broker对应的kafka进程，观察kafka是否能够选举出来新的Controller。



3.2.4 leader是否均匀分配

启动Kafka集群时选举leader

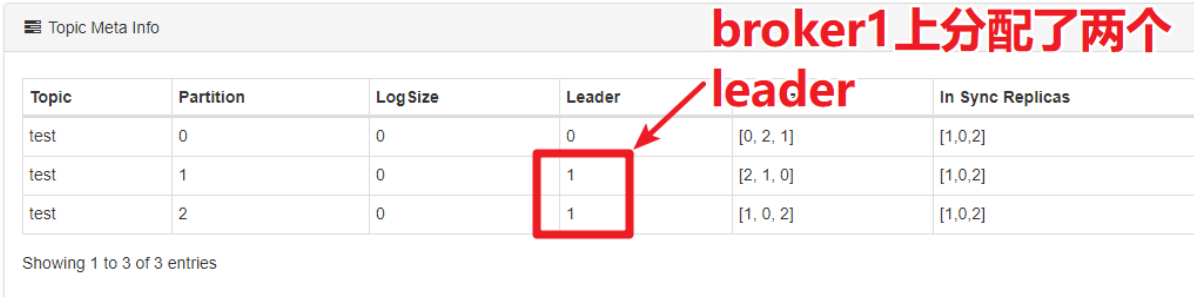
Kafka中引入了一个叫做「preferred-replica」的概念，意识就是：优先的Replica。在ISR列表中，第一个replica就是preferred-replica。第一个分区存放的broker，肯定就是preferred-replica。

执行以下脚本可以将preferred-replica设置为leader。均匀分配每个分区的leader。

```
./kafka-leader-election.sh --bootstrap-server node1.itcast.cn:9092 --topic 主题 --partition=1 --election-type preferred
```

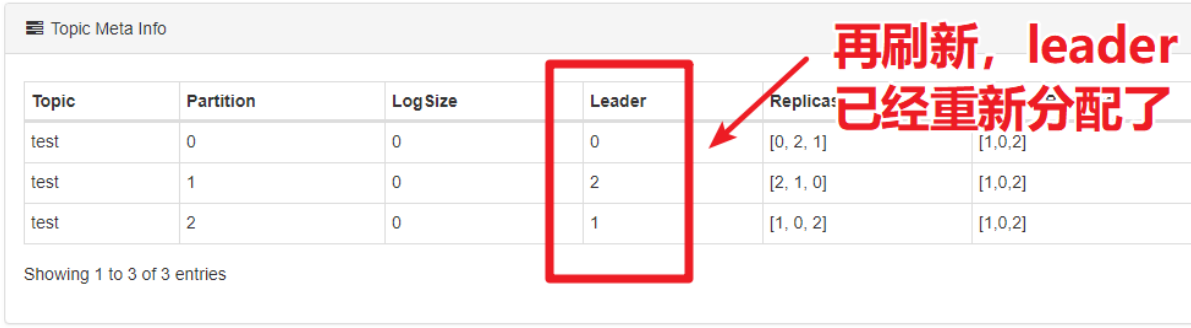
试一试

杀掉test主题的某个broker，这样kafka会重新分配leader。等到Kafka重新分配leader之后，再次启动kafka进程。此时：观察test主题各个分区leader的分配情况。



此时，会造成leader分配是不均匀的，所以可以执行以下脚本来重新分配leader:

```
./kafka-leader-election.sh --bootstrap-server node1.itcast.cn:9092 --topic test --partition=1 --election-type preferred
```



3.2.5 某个leader出现故障/更新分区副本数量时

1. Controller发现某个分区的leader崩溃

2. 如果该分区的ISR中还存在Replica存活，则选择其中一个作为新的leader，新的ISR中，包含当前ISR中所有幸存的Replica

3. 如果没有任何Replica存活，则该分区的任意的Replica作为新的leader以及ISR

4. 如果分区的所有Replica都已经宕机，则将新的leader设置为1

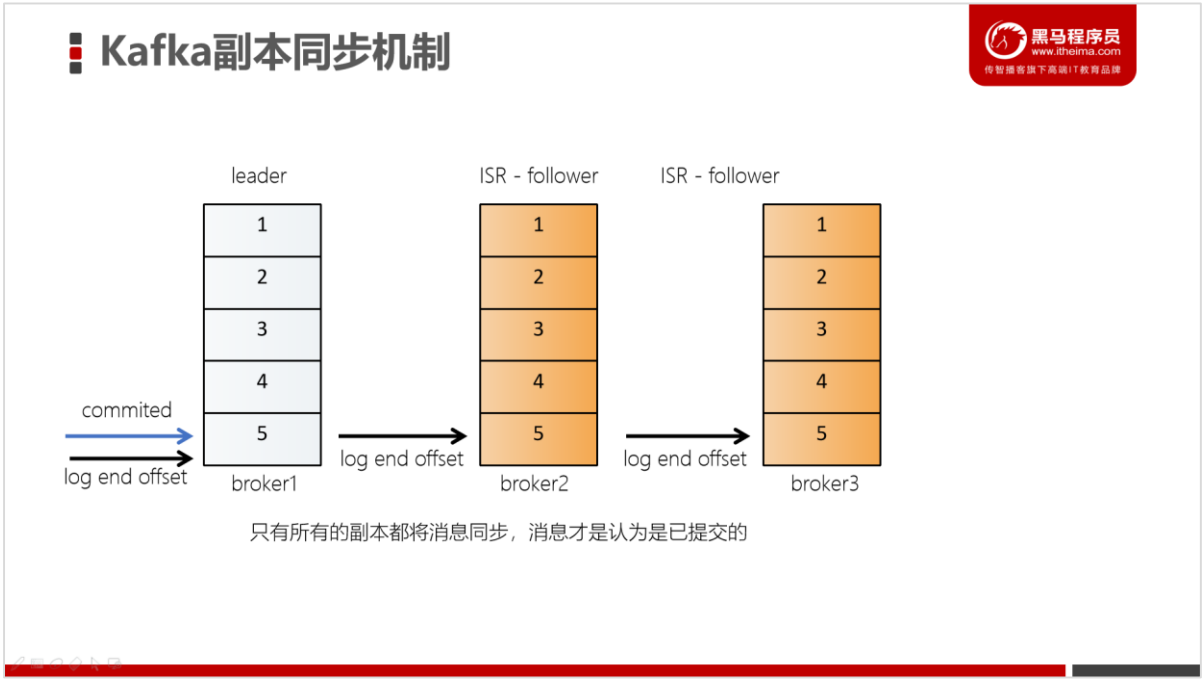
Kafka的这种选择leader的方式，非常迅速。

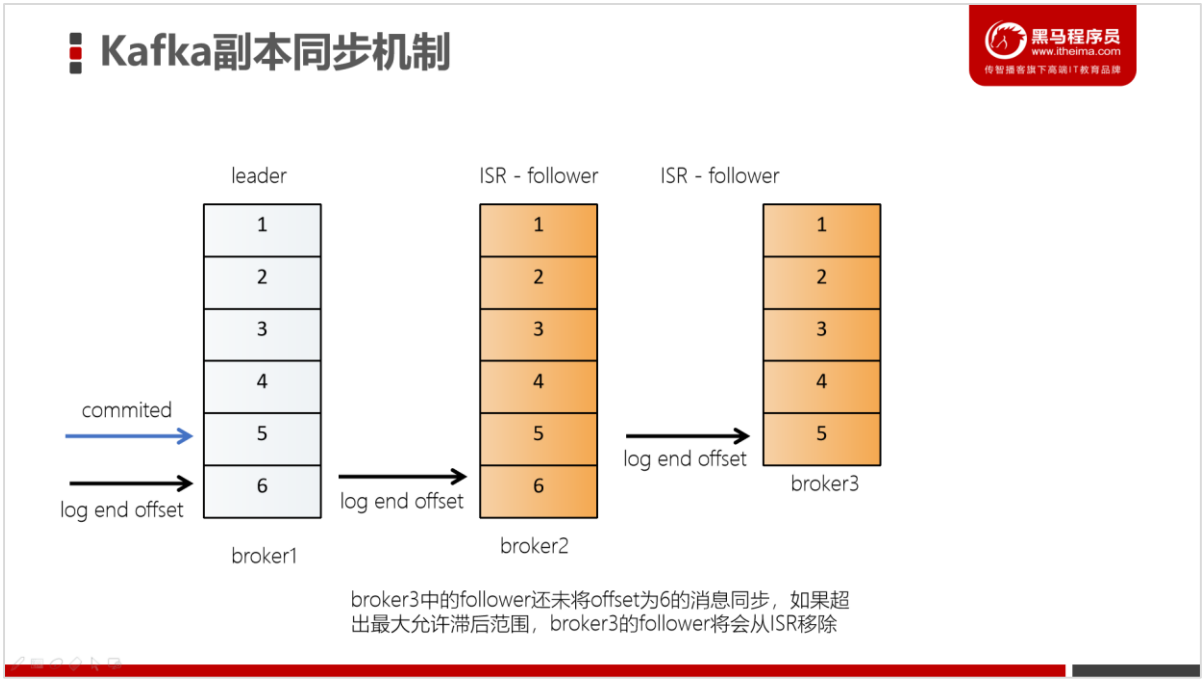
3.3 Kafka副本同步机制

3.3.1 Kafka 0.11以前

Kafka的副本同步机制是建立在ISR的基础上的。一条新的消息，只有被ISR中的副本都接收到，才被视为「已同步」状态。这与ZK的同步机制不一样，ZK只要超过半数节点写入，就认为是已写入成功。

Kafka分区的follower会定期从leader中同步数据。

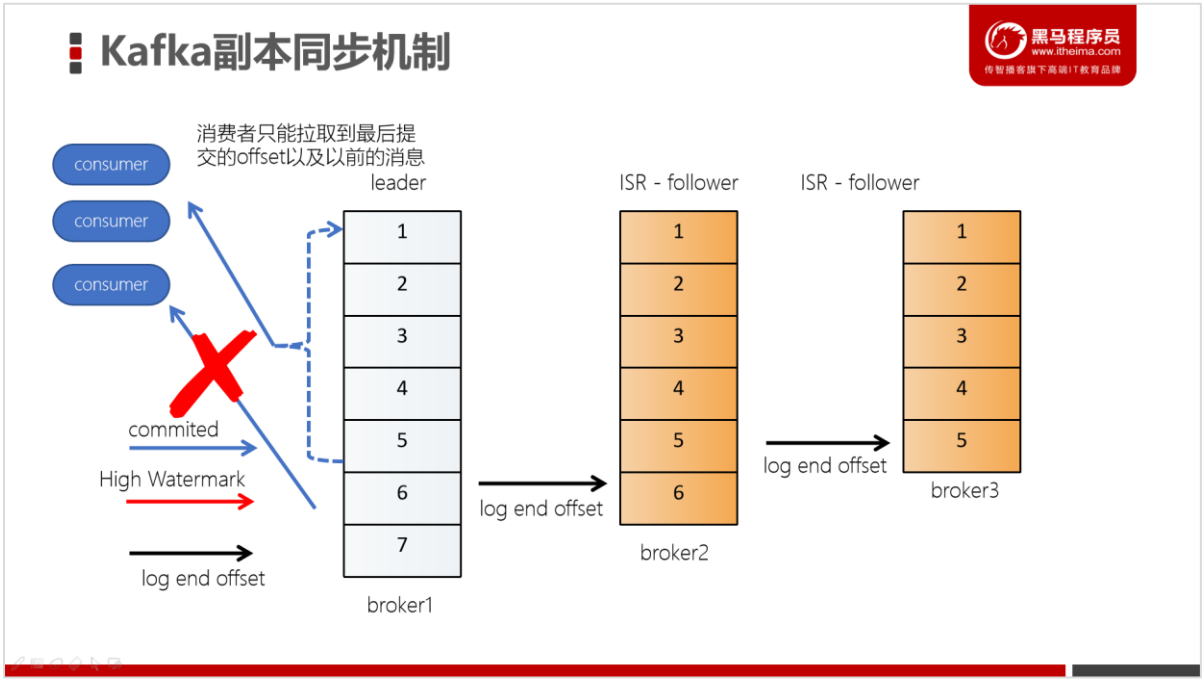




以下两个配置，可以配置replica同步leader消息时，最大允许的滞后的配置：

replica.lag.max.messages：最大允许落后多少条消息（根据offset来判断）

replica.lag.time.max：最大允许落后多长时间（只要replica发送给leader的同步请求时间超过这个数字，就会认为是滞后的，leader将会从ISR中将此replica移除）



HW（High Watermark）：高水位，消费者只能拉取到这个offset之前的消息。上图中的HW为6。也就是说，消费者只能消费到6（HW）以前的offset消息。

LEO（Log End Offset）：副本的最后一条消息的offset

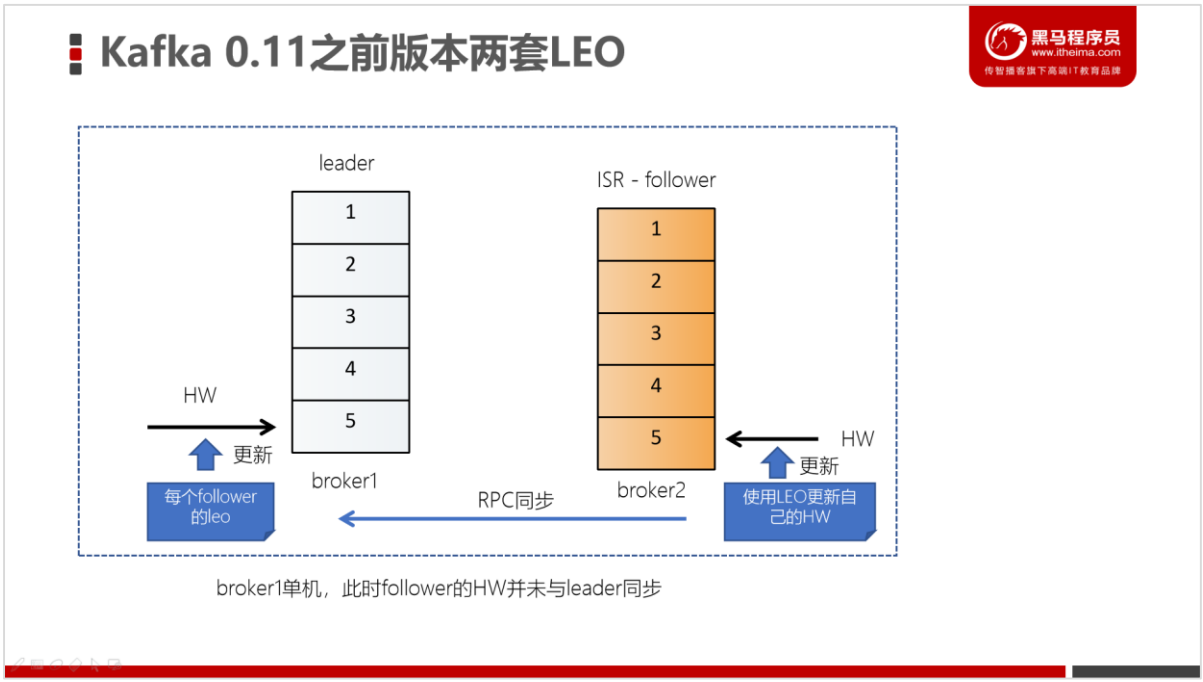
3.3.2 0.11之前版本的问题

follower需要不停地向leader发送同步请求，而实际上：一个follower的LEO会在两个地方同时维护。

1. 一套LEO保存在follower副本所在broker中

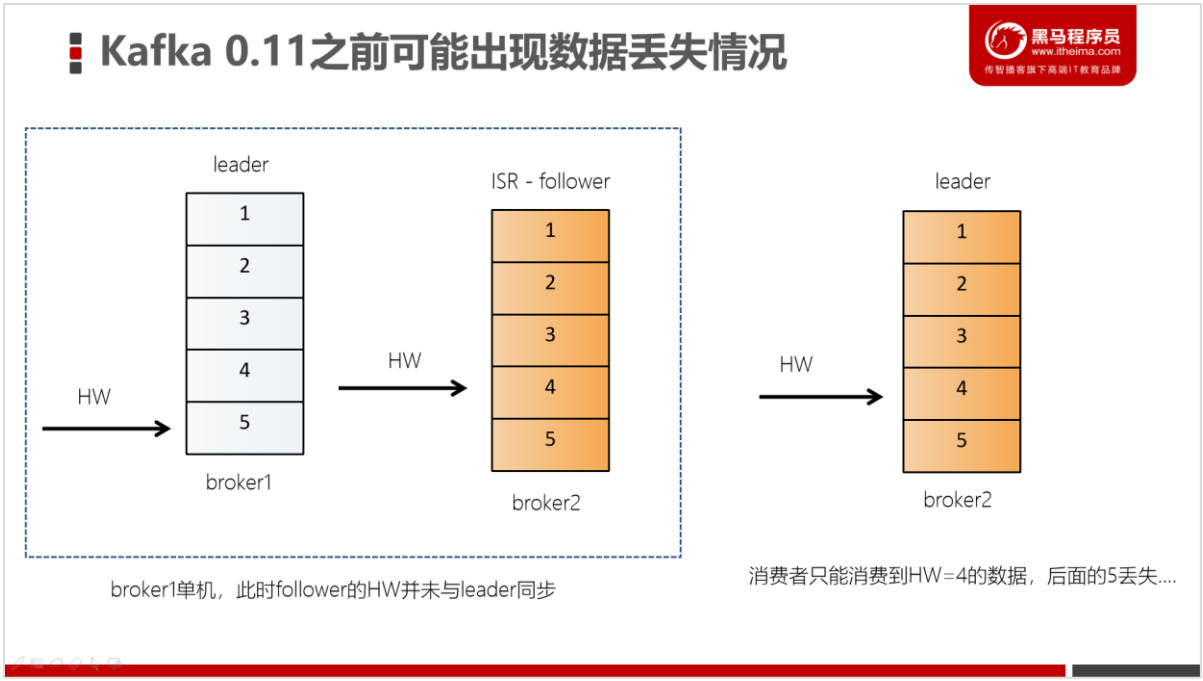
2. 另一套LEO保存在leader副本所在broker中——换句话说，leader副本机器上保存了所有的follower副本的LEO。

Kafka利用第一套LEO更新follower副本的HW值，使用另一套LEO更新leader的HW值。



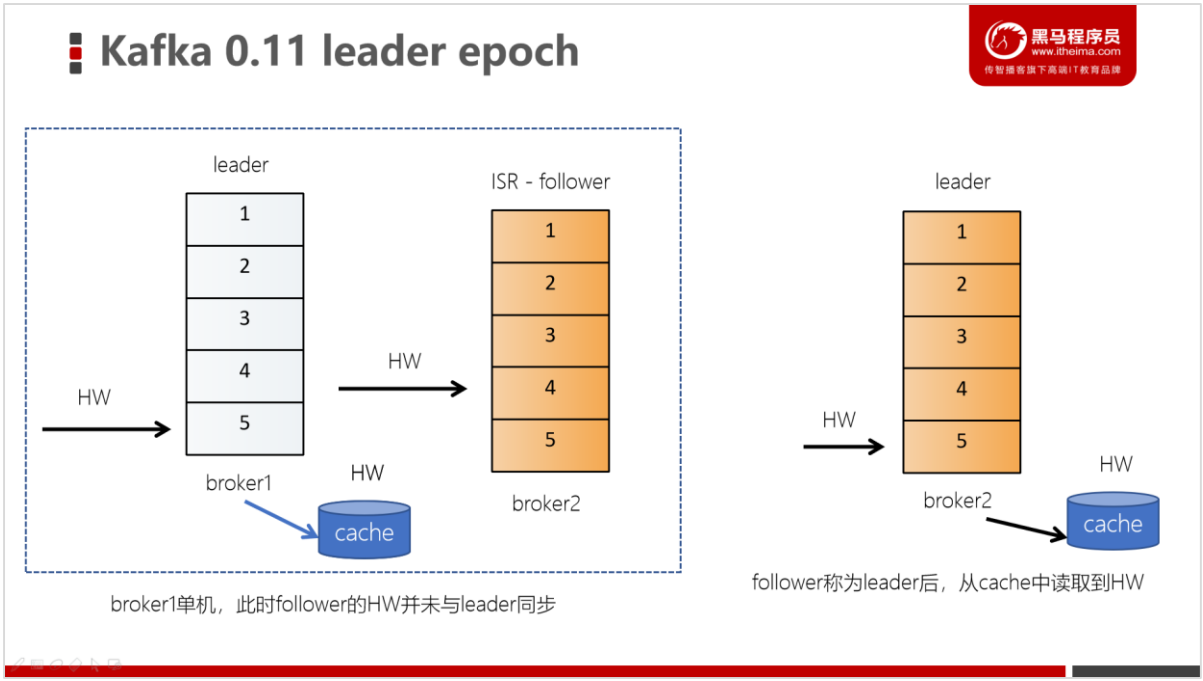
PS：因为每个follower都可能会成为leader，所以每个副本都必须维护HW和LEO。

Kafka使用HW值来决定副本备份的进度，而HW值的更新通常需要额外一轮FETCH RPC才能完成，故而这种设计是有问题的。如果两套LEO不同步，会导致HW不一致，从而导致数据丢失。



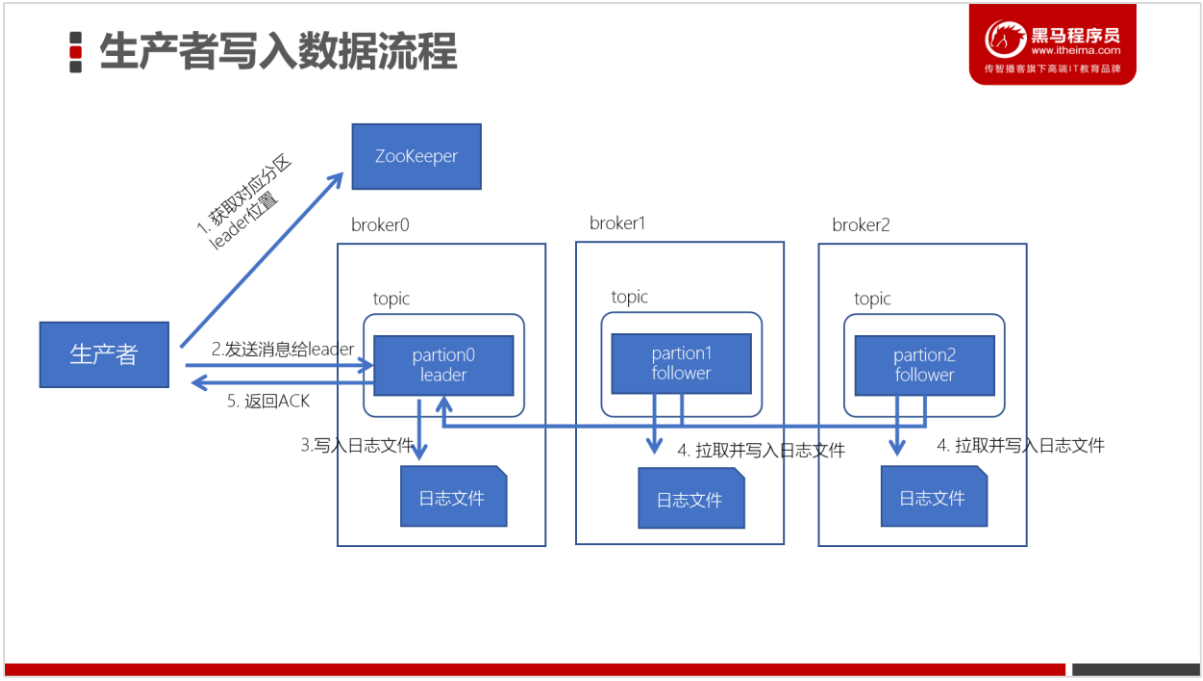
3.3.3 leader epoch机制

Kafka从0.11开始，引入leader epoch机制。该机制将LEO保存到一个缓存中，该缓存会定期保存到checkpoint文件中。当leader宕机后，新的follower称为leader，会从该缓存中读取LEO，这样可以确保数据不丢失。

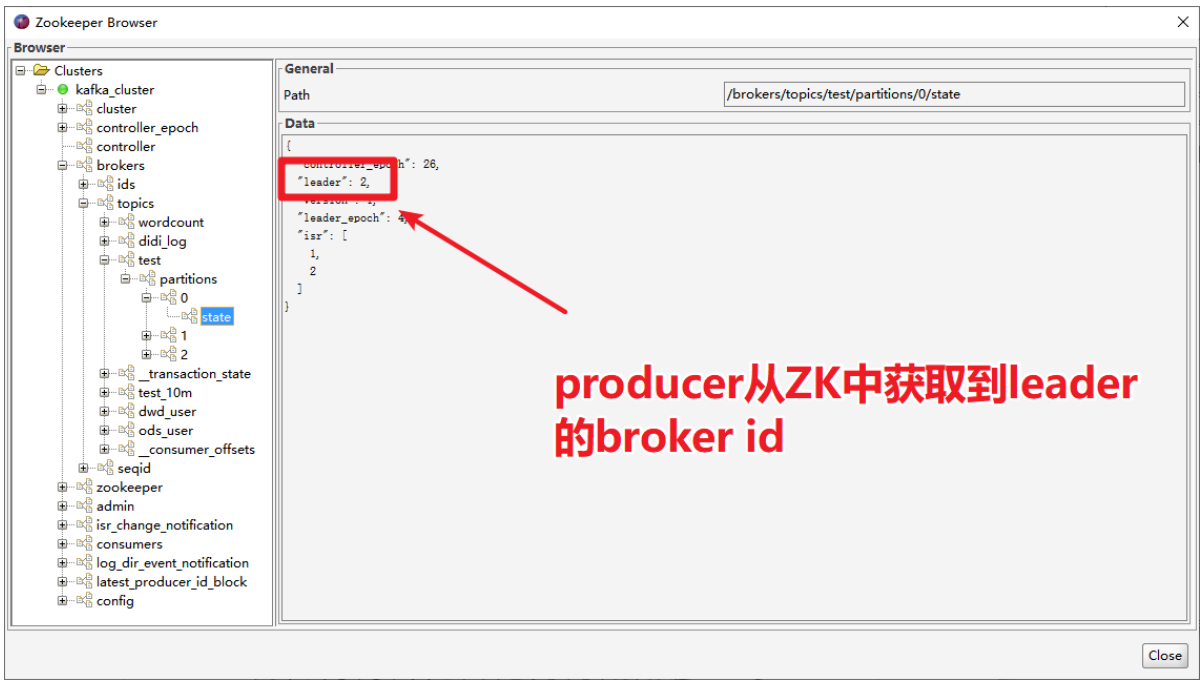


3.4 Kafka生产、消费数据工作流程

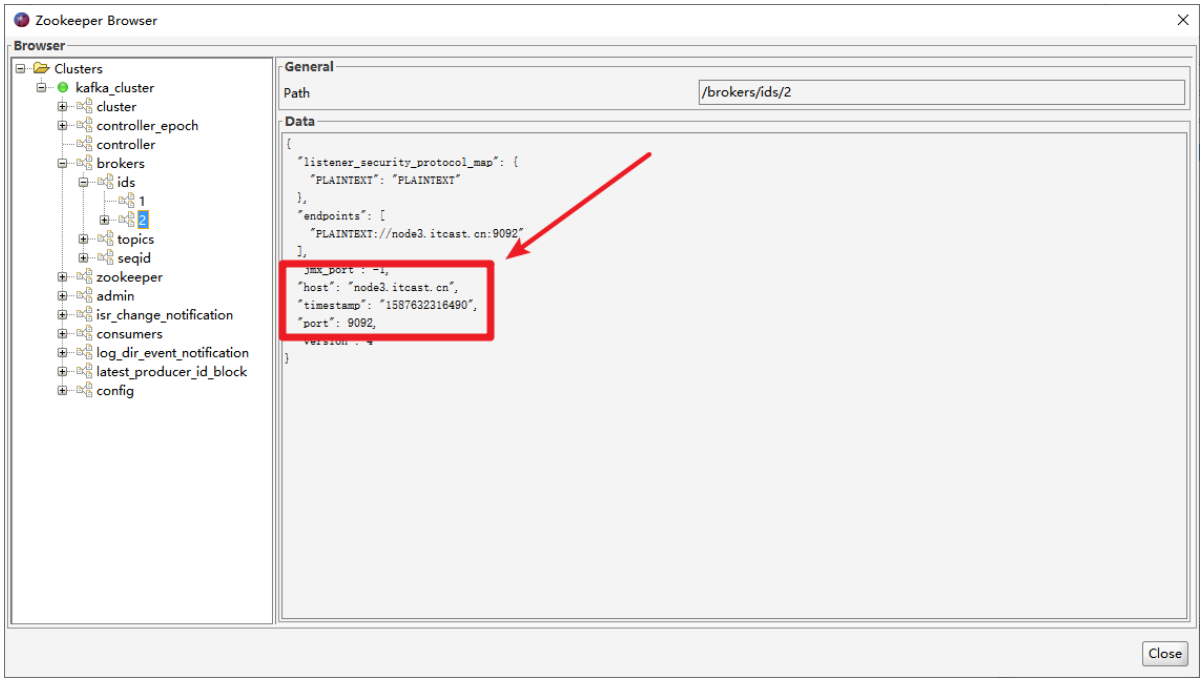
3.4.1 Kafka数据写入流程



- 生产者先从 zookeeper 的 "/brokers/topics/主题名/partitions/分区名/state"节点找到该 partition 的leader



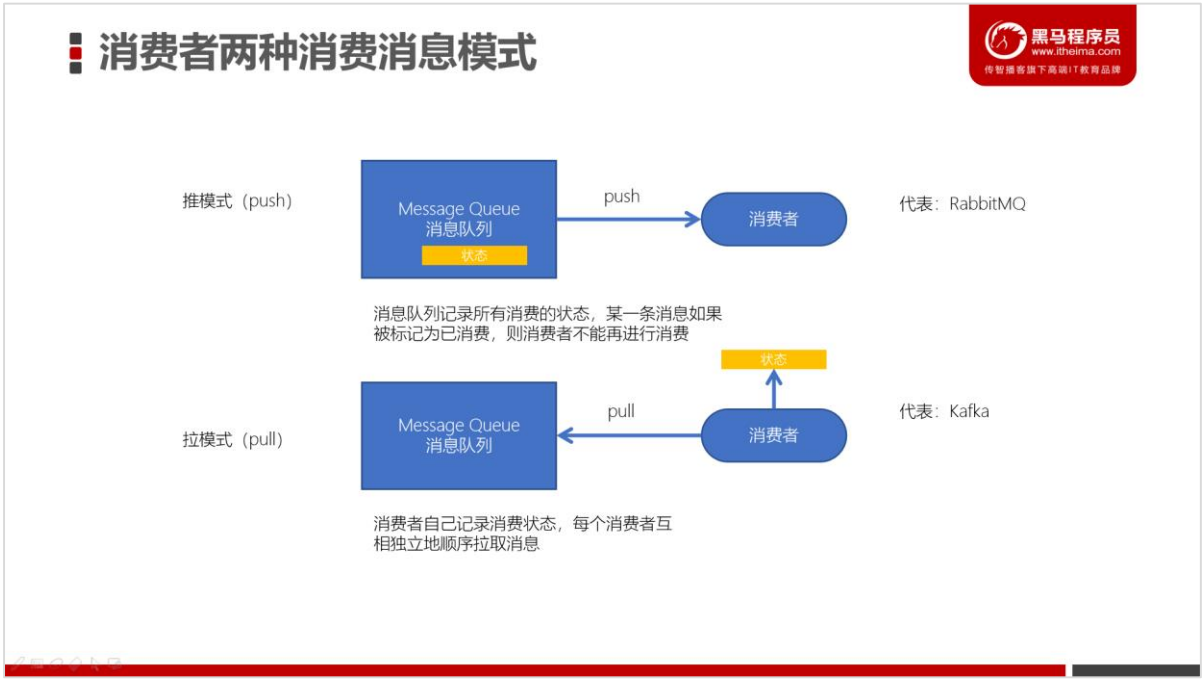
- 生产者在ZK中找到该ID找到对应的broker



- broker进程上的leader将消息写入到本地log中
- follower从leader上拉取消息，写入到本地log，并向leader发送ACK
- leader接收到所有的ISR中的Replica的ACK后，增加HW（高水位——表示消费者能够消费到的位置），并向生产者返回ACK。

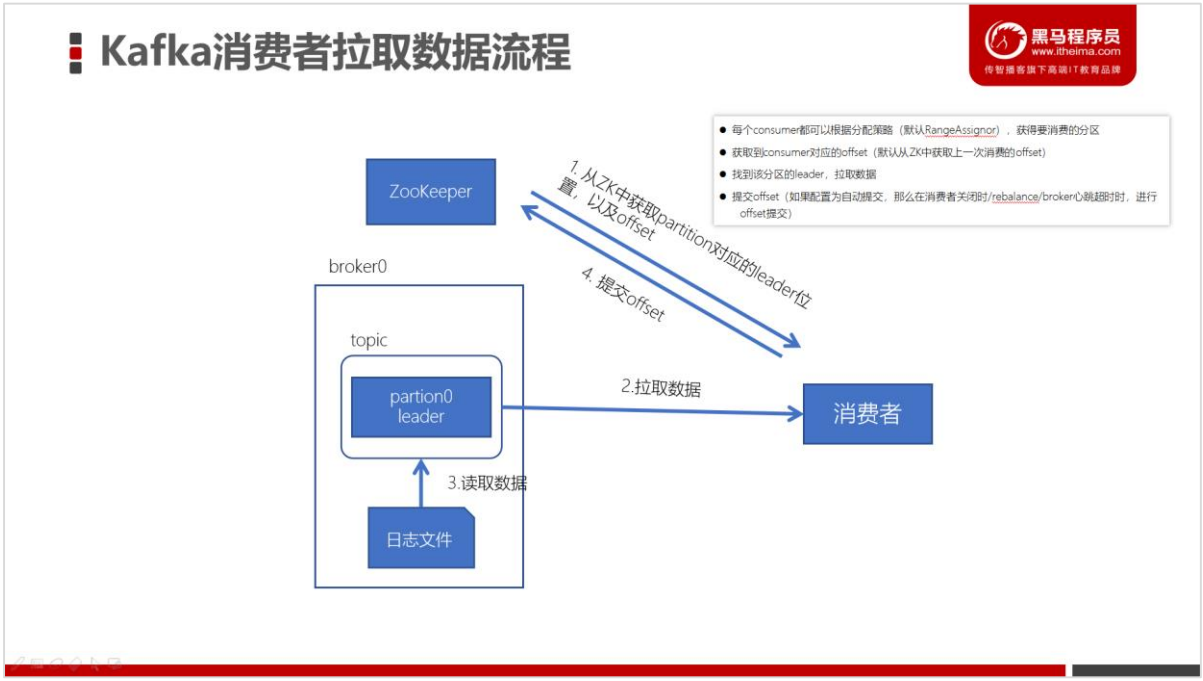
3.4.2 Kafka数据消费流程

3.4.2.1 两种消费模式



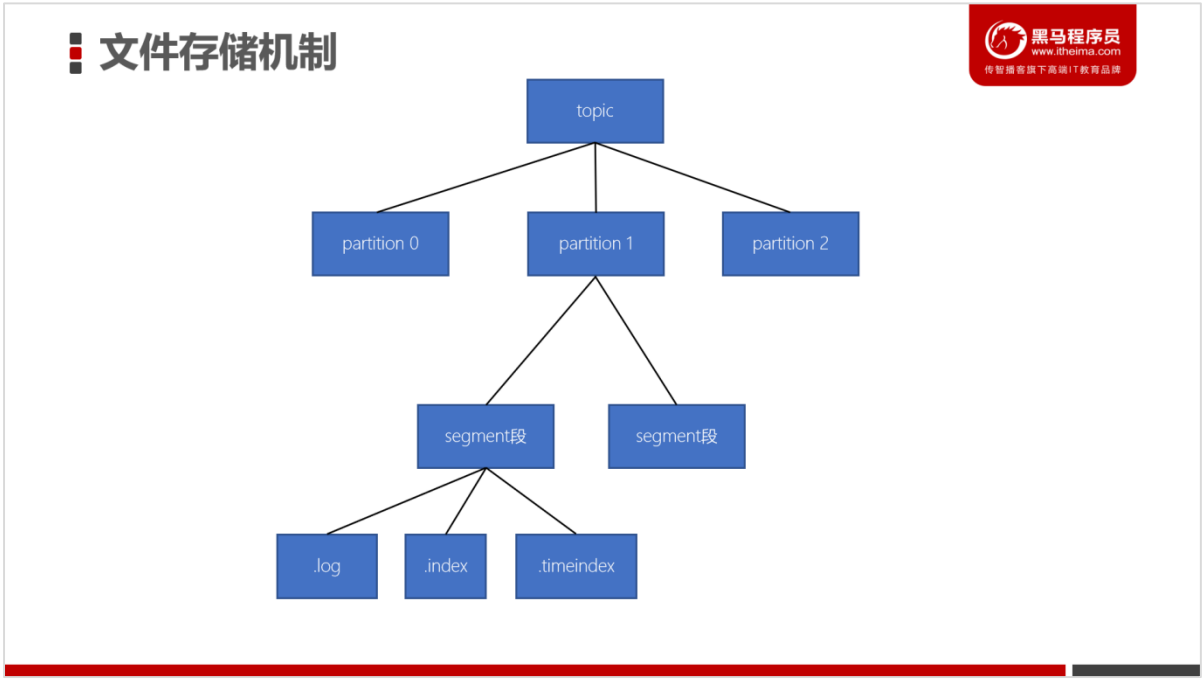
- kafka采用拉取模型，由消费者自己记录消费状态，每个消费者互相独立地顺序拉取每个分区的消息
- 消费者可以按照任意的顺序消费消息。比如，消费者可以重置到旧的偏移量，重新处理之前已经消费过的消息；或者直接跳到最近的位置，从当前的时刻开始消费。

3.4.2.2 Kafka消费数据流程



- 每个consumer都可以根据分配策略（默认RangeAssignor），获得要消费的分区
- 获取到consumer对应的offset（默认从ZK中获取上一次消费的offset）
- 找到该分区的leader，拉取数据
- 提交offset（如果配置为自动提交，那么在消费者关闭时/rebalance/broker心跳超时时，进行offset提交）

3.5 Kafka的数据存储形式



- 一个topic由多个分区组成
- 一个分区（partition）由多个segment（段）组成
- 一个segment（段）由多个文件组成

3.5.1 存储日志

接下来，我们来看一下Kafka中的数据到底是如何在磁盘中存储的。Kafka中的数据是保存在/export/server/kafka_2.12-2.4.1/data中。消息是保存在以：「主题名-分区ID」的文件夹中的。

该文件夹中包含以下内容：

```
-rw-r--r--. 1 root root 10M 4月    9 23:50 00000000000000000000.index
-rw-r--r--. 1 root root 11M 4月    9 23:50 00000000000000000000.log
-rw-r--r--. 1 root root 10M 4月    9 23:50 00000000000000000000.timeindex
-rw-r--r--. 1 root root    8 4月    9 23:43 leader-epoch-checkpoint
```

这些分别对应：

文件名	说明
00000000000000000000.index	索引文件，根据offset查找数据就是通过该索引文件来操作的

00000000000000000000.log	日志数据文件
00000000000000000000.timeindex	时间索引文件
leader-epoch-checkpoint	持久化每个partition leader对应的LEO

- 每个日志文件的文件名为起始偏移量，因为每个分区的起始偏移量是0，所以，分区的日志文件都以00000000000000000000.log开始
- 默认的每个日志文件最大为「log.segment.bytes =1024*1024*1024」1G
- 为了简化根据offset查找消息，Kafka日志文件名设计为开始的偏移量

3.5.1.1 观察测试

为了方便测试观察，新创建一个topic：「test_10m」，该topic每个日志数据文件最大为10M

```
bin/kafka-topics.sh --create --zookeeper node1.itcast.cn --topic test_10m --replication-factor 2 --partitions 3 --config segment.bytes=10485760
```

使用之前的生产者程序往「test_10m」主题中生产数据，可以观察到如下：

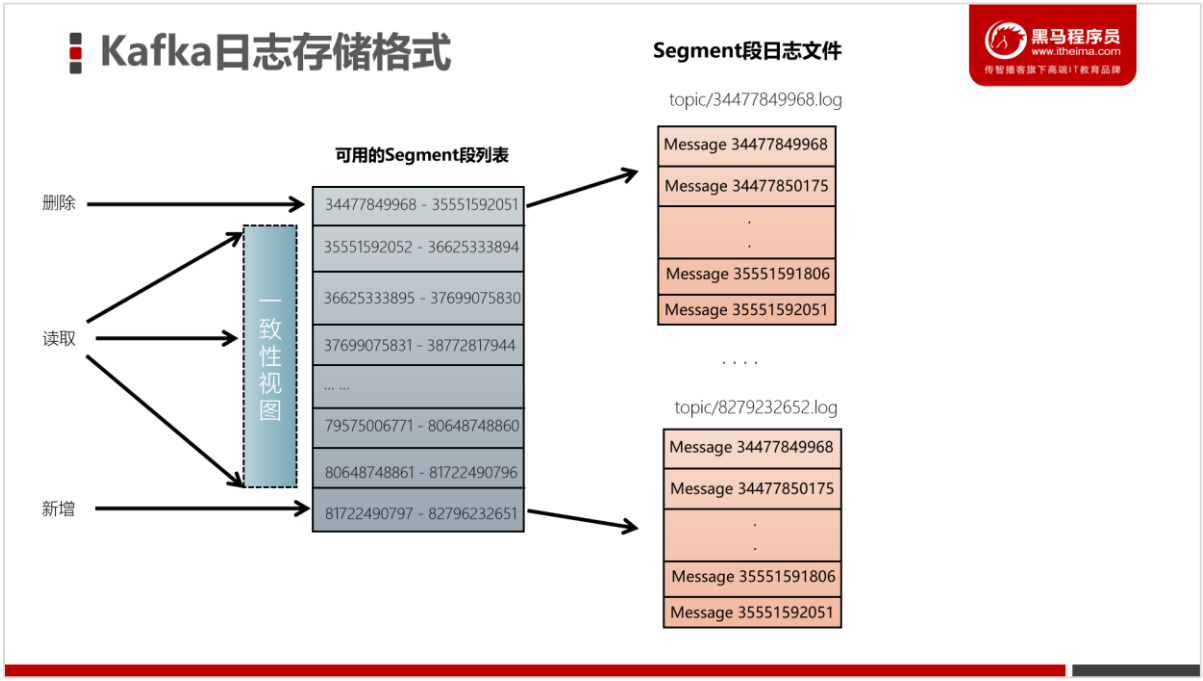
```
-rw-r--r--. 1 root root 5.1K 4月  10 00:26 00000000000000000000.index
-rw-r--r--. 1 root root  10M 4月  10 00:26 00000000000000000000.log
-rw-r--r--. 1 root root 7.5K 4月  10 00:26 00000000000000000000.timeindex
-rw-r--r--. 1 root root 5.0K 4月  10 00:26 00000000000000435086.index
-rw-r--r--. 1 root root  10M 4月  10 00:26 00000000000000435086.log
-rw-r--r--. 1 root root   10 4月  10 00:26 00000000000000435086.snapshot
-rw-r--r--. 1 root root 7.5K 4月  10 00:26 00000000000000435086.timeindex
-rw-r--r--. 1 root root 5.0K 4月  10 00:27 00000000000000869646.index
-rw-r--r--. 1 root root  10M 4月  10 00:27 00000000000000869646.log
-rw-r--r--. 1 root root   10 4月  10 00:26 00000000000000869646.snapshot
-rw-r--r--. 1 root root 7.5K 4月  10 00:27 00000000000000869646.timeindex
-rw-r--r--. 1 root root 5.0K 4月  10 00:27 00000000000001304206.index
-rw-r--r--. 1 root root  10M 4月  10 00:27 00000000000001304206.log
-rw-r--r--. 1 root root   10 4月  10 00:27 00000000000001304206.snapshot
-rw-r--r--. 1 root root 7.5K 4月  10 00:27 00000000000001304206.timeindex
-rw-r--r--. 1 root root  10M 4月  10 00:27 00000000000001738766.index
-rw-r--r--. 1 root root 9.3M 4月  10 00:27 00000000000001738766.log
-rw-r--r--. 1 root root   10 4月  10 00:27 00000000000001738766.snapshot
-rw-r--r--. 1 root root  10M 4月  10 00:27 00000000000001738766.timeindex
-rw-r--r--. 1 root root    8 4月  10 00:26 leader-epoch-checkpoint
```

每个log文件的大小最大为 10M

3.5.1.2 写入消息

- 新的消息总是写入到最后的一个日志文件中
- 该文件如果到达指定的大小（默认为：1GB）时，将滚动到一个新的文件中

3.5.1.3 读取消息



- Kafka读取消息主要就是根据「offset」，以及每个日志文件的最大大小来读取消息的
- 根据 「offset」首先需要找到存储数据的 segment 段（注意：offset指定分区的全局偏移量）
- 然后根据这个「全局分区offset」找到相对于文件的「segment段offset」
- 最后再根据 「segment段offset」读取消息
- 为了提高查询效率，每个文件都会维护对应的范围内存，查找的时候就是使用简单的二分查找

3.5.1.4 删除消息

- 在Kafka中，消息是会被**定期清理**的。一次删除一个segment段的日志文件
- Kafka的日志管理器，会根据Kafka的配置，来决定哪些文件可以被删除

3.6 消息不丢失机制

3.6.1 broker数据不丢失

生产者通过分区的leader写入数据后，所有在ISR中follower都会从leader中复制数据，这样，可以确保即使leader崩溃了，其他的follower的数据仍然是可用的

3.6.2 生产者数据不丢失

- 生产者连接leader写入数据时，可以通过ACK机制来确保数据是已经成功写入的。ACK机制有三个可选配置
  1. 配置ACK响应要求为 -1 时 —— 表示所有的节点都收到数据(leader和follower都接收到数据）
  2. 配置ACK响应要求为 1 时 —— 表示leader收到数据
  3. 配置ACK影响要求为 0 时 —— 生产者只负责发送数据，不关心数据是否丢失（这种情况可能会产生数据丢失，但性能是最好的）
- 生产者可以采用同步和异步两种方式发送数据
  - 同步：发送一批数据给kafka后，等待kafka返回结果
  - 异步：发送一批数据给kafka，只是提供一个回调函数。

说明：如果broker迟迟不给ack，而buffer又满了，开发者可以设置是否直接清空buffer中的数1据。

3.6.3 消费者数据不丢失

在消费者消费数据的时候，只要每个消费者记录好offset值即可，就能保证数据不丢失。

4. Kafka中数据清理（Log Deletion）

Kafka的消息存储在磁盘中，为了控制磁盘占用空间，Kafka需要不断地对过去的一些消息进行清理工作。Kafka的每个分区都有很多的日志文件，这样也是为了方便进行日志的清理。在Kafka中，提供两种日志清理方式：

- 日志删除（Log Deletion）：按照指定的策略**直接删除**不符合条件的日志。
- 日志压缩（Log Compaction）：按照消息的key进行整合，有相同key的但有不同value值，只保留最后一个版本。

在Kafka的broker或topic配置中：

配置项	配置值	说明
log.cleaner.enable	true（默认）	开启自动清理日志功能
log.cleanup.policy	delete（默认）	删除日志

log.cleanup.policy	compact	压缩日志
log.cleanup.policy	delete,compact	同时支持删除、压缩

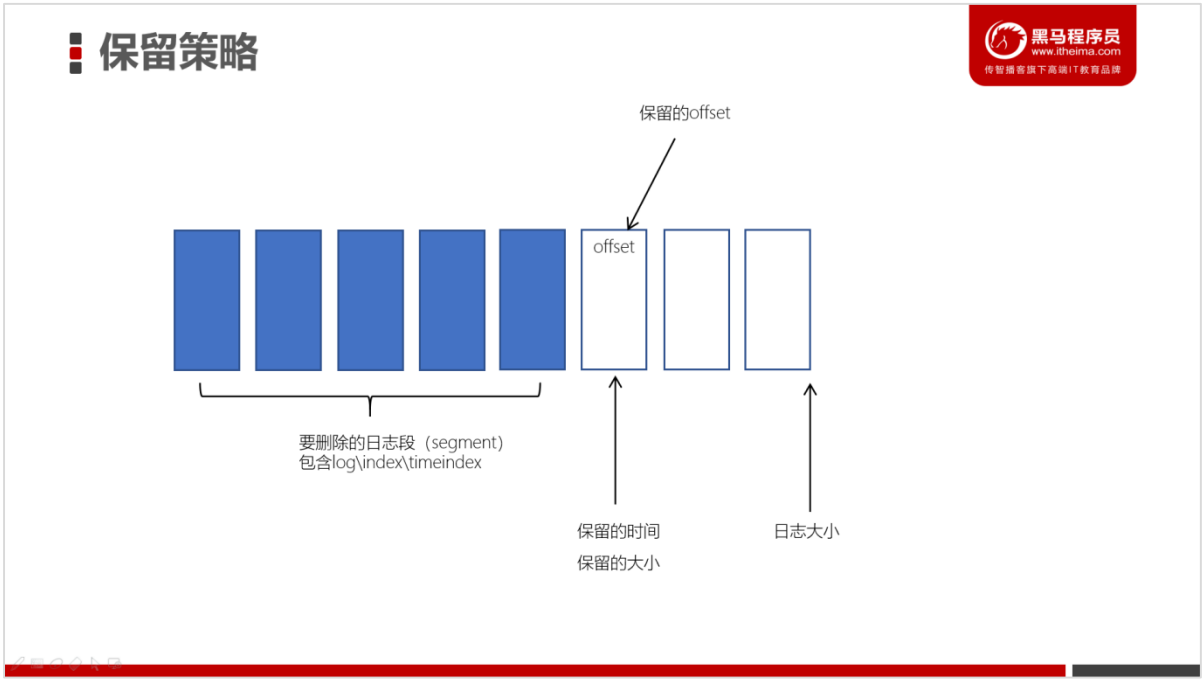
4.1 日志删除

日志删除是以段（segment日志）为单位来进行定期清理的。

4.1.1 定时日志删除任务

Kafka日志管理器中会有一个专门的日志删除任务来定期检测和删除不符合保留条件的日志分段文件，这个周期可以通过broker端参数log.retention.check.interval.ms来配置，默认值为300,000，即5分钟。当前日志分段的保留策略有3种：

1. 基于时间的保留策略
2. 基于日志大小的保留策略
3. 基于日志起始偏移量的保留策略



4.1.2 基于时间的保留策略

以下三种配置可以指定如果Kafka中的消息超过指定的阈值，就会将日志进行自动清理：

- log.retention.hours

- log.retention.minutes
- log.retention.ms

其中，优先级为 log.retention.ms > log.retention.minutes > log.retention.hours。默认情况，在broker中，配置如下：

log.retention.hours=168

也就是，默认日志的保留时间为168小时，相当于保留7天。

删除日志分段时:

1. 从日志文件对象中所维护日志分段的跳跃表中移除待删除的日志分段，以保证没有线程对这些日志分段进行读取操作

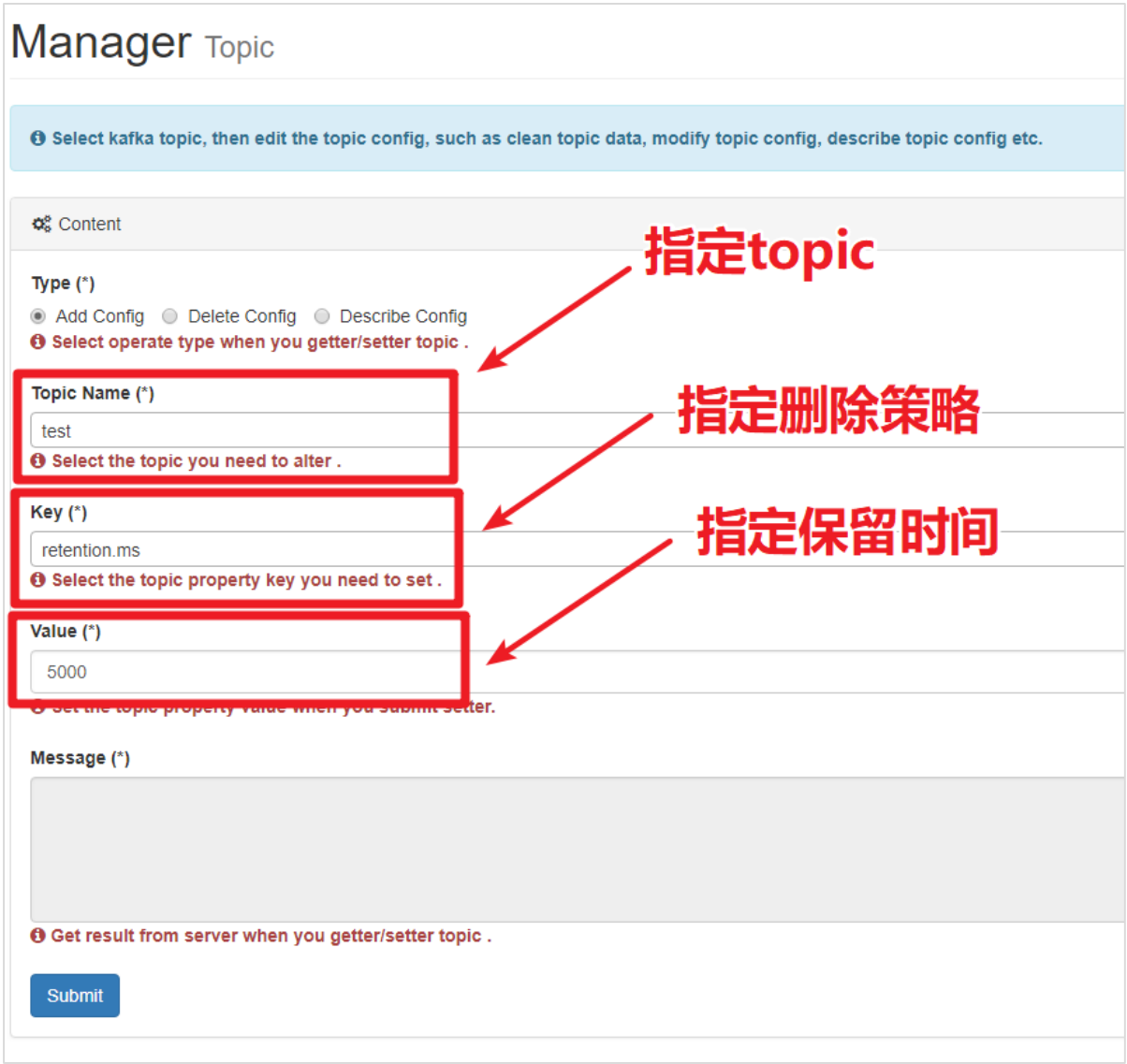
2. 将日志分段文件添加上".deleted"的后缀（也包括日志分段对应的索引文件）

3. Kafka的后台定时任务会定期删除这些".deleted"为后缀的文件，这个任务的延迟执行时间可以通过file.delete.delay.ms参数来设置，默认值为60000，即1分钟。

4.1.2.1 试一试：设置topic 5秒删除一次

1. 为了方便观察，设置段文件的大小为1M。

2. 设置topic的删除策略



尝试往topic中添加一些数据，等待一会，观察日志的删除情况。我们发现，日志会定期被标记为删除，然后被删除。

4.1.3 基于日志大小的保留策略

日志删除任务会检查当前日志的大小是否超过设定的阈值来寻找可删除的日志分段的文件集合。可以通过broker端参数 log.retention.bytes 来配置，默认值为-1，表示无穷大。如果超过该大小，会自动将超出部分删除。

注意:

log.retention.bytes 配置的是日志文件的总大小，而不是单个的日志分段的大小，一个日志文件包含多个日志分段。

4.1.4 基于日志起始偏移量保留策略

每个segment日志都有它的起始偏移量，如果起始偏移量小于 logStartOffset，那么这些日志文件将会标记为删除。

4.2 日志压缩

Log Compaction会生成新的日志分段文件，日志分段中每条消息的物理位置会重新按照新文件来组织。Log Compaction执行过后的偏移量不再是连续的，不过这并不影响日志的查询。

5. Kafka为什么这么快？

Kafka的消息是保存或缓存在磁盘上的，一般认为在磁盘上读写数据是会降低性能的，因为寻址会比较消耗时间，但是实际上，Kafka的特性之一就是高吞吐率。即使是普通的服务器，Kafka也可以轻松支持每秒百万级的写入请求，超过了大部分的消息中间件。那为什么Kafka这么快呢？

5.1 生产数据

Kafka生产数据很快的原因，主要有两个：

1. 顺序写入

2. MMFile

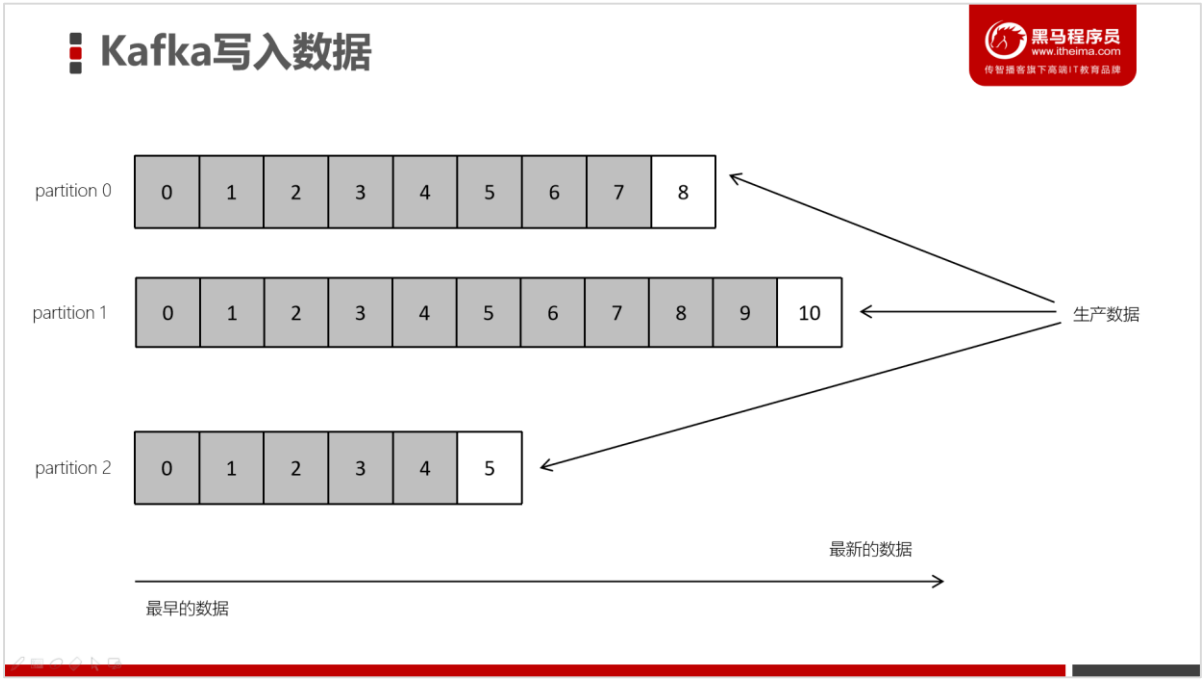
5.1.1 顺序写入

5.1.1.1 顺序读写与随机读写

磁盘读写的快慢取是**顺序读写**或者**随机读写**。在顺序读写的情况下，某些优化场景磁盘的读写速度可以和内存持平。因为硬盘是机械结构，每次读写都会寻址->写入，其中寻址是一个“机械动作”，它是最耗时的。为了提高读写硬盘的速度，Kafka就是使用顺序I/O。

Linux对于磁盘的读写也有优化，包括read-ahead和write-behind，磁盘缓存等。如果在内存做这些操作的时候，一个是JAVA对象的内存开销很大，另一个是随着堆内存数据的增多，JAVA的GC时间会变得很长，使用磁盘操作有以下几个好处：

- 磁盘顺序读写速度超过内存**随机读写**
- JVM的GC效率低，内存占用大。使用磁盘可以避免这一问题



上图就展示了Kafka是如何写入数据的， Kafka会把数据插入到文件末尾（白色部分）。

这种方法没有办法删除数据 ，Kafka会把所有的数据都保留下来，每个消费者（Consumer）对每个Topic都有一个offset用来表示 读取到了第几条数据。只有日志管理任务会定期清理过期的段日志文件。

5.1.1.2 Memory Mapped Files

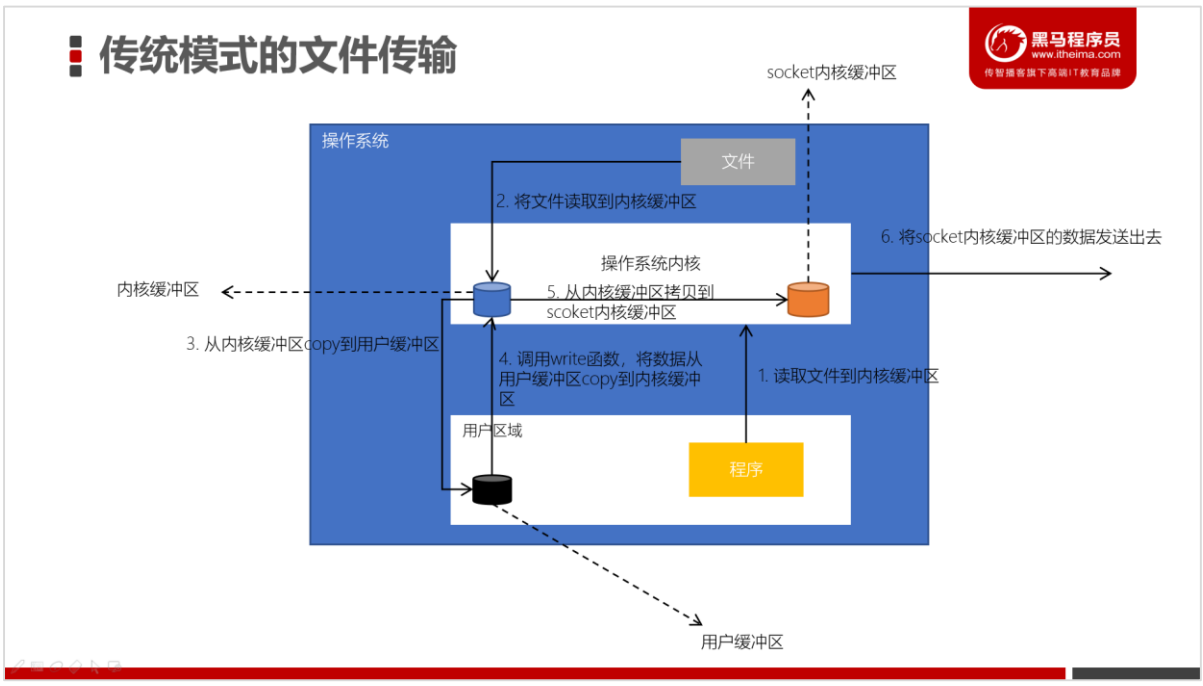
即便是顺序写入硬盘，硬盘的速度还是不可能追上内存。Kafka的数据并不是实时的写入硬盘 ，它充分利用了现代操作系统**分页存储**来利用内存提高I/O效率。

Memory Mapped Files(后面简称mmap)也被翻译成内存映射文件，在64位操作系统中一般可以表示20G的数据文件，它的工作原理是直接利用操作系统的Page来实现文件到物理内存的直接映射。完成映射之后你对物理内存的操作会被同步到硬盘上（操作系统在适当的时候）。

通过mmap，进程像读写硬盘一样读写内存（当然是虚拟机内存），也不必关心内存的大小，因为还有虚拟内存。使用这种方式可以获取很大的I/O提升。

5.2 消费数据

5.2.1 传统模式的数据传输



5.2.2 Kafka使用sendfile来实现零拷贝