Kafka□理文档

Spring Kafka

Kafka官方文档

□述

Kafka偏重吞吐与持久化

定义：Kafka是一个分布式的基于发布/订阅模式的消息队列，主要应用于大数据实时处理领域

发布/订阅：消息的发布者不会将消息直接发布给特定的订阅者，而是将发布的消息分为不同的类别，订阅者只接收感兴趣的消息（和RabbitMQ一样）

传统消息队列的应用场景

传统的消息队列的主要应用场景包括：缓存/消峰、解耦和异步通信

缓冲/削峰：有助于控制和优化数据流经过系统的速度，解决生产消息和消费消息的处理速度不一致的情况

image-20251023233518657

解耦：允许你独立的扩展或修改两边的处理过程，只要确保它们遵守同样的接口约束

image-20251023233533002

异步通信：允许用户把一个消息放入队列，但并不立即处理它，然后再需要的时候再去处理它们

image-20251023233546291

Kafka基础架构

整体上由

- `Producer`
- `Consumer`
- `Kakfa Cluster`

组成

image-20251023233656992

- `Producer`：消息生产者，就是向`Kafka broker`发消息的客户端
- `Consumer`：消息消费者，向`Kafka broker`取消息的客户端
- `Consumer Group(CG)`：消费者组，由多个`consumer`组成
  - 消费者组内每个消费者负责消费不同分区的数据
  - 消费者组之间互不影响，所有的消费者都属于某个消费者组

image-20251025101503778

- `Broker`：**一台`Kafka`服务器就是一个`broker`**，一个集群由多个`broker`组成，一个`broker`可以容纳多个`topic`

- `Topic`：可以理解为一个队列，生产者和消费者面向的都是一个 `topic`
- `Partition`：分区，一个非常大的 `Topic` 可以分布到多个 `broker` 上，也就是每个 `Topic` 会被分成多个 `Partition`，提高并行度和可扩展性

image-20251025101859263

- `Replica`：副本，一个 `topic` 的每个分区都有若干个副本，一个 `leader` 和若干个 `follower`
- `Leader`：`Partition` 的主副本，负责读写
- `Follower`：`Partition` 的从副本，负责同步

Kafka API

接下来只阐述了在**Spring**项目中如何使用**Kafka**

主要组件

- `KafkaTemplate`：用于发送消息
- `@KafkaListener`：用于消费消息

下面的组件主要用于高级自定义和底层的了解

- `ProducerFactory`：创建生产者实例
- `ConsumerFactory`：创建消费者实例，负责连接配置
- `ConcurrentKafkaListenerContainerFactory`：控制监听线程数量、并发消费等
- `KafkaProducer`：真正和Kafka集群通信的组件，`KafkaTemplate` 底层委托这个组件进行消息发送
- `DefaultKafkaProducerFactory`：是 `ProducerFactory` 的实现类，用于加载配置并创建 `KafkaProducer` 实例
- `ContainerFactory`：用于创建和管理消息监听容器（`MessageListenerContainer`）的工厂，负责消费行为配置

生产者

使用步骤

- 引入依赖

```xml
<dependency>
  <groupId>org.springframework.kafka</groupId>
  <artifactId>spring-kafka</artifactId>
  <version>3.3.10</version>
</dependency>
```

- 生产者配置
    - 在yaml文件进行生产者的配置

```yaml
spring:
  kafka:
    bootstrap-servers: localhost:9092  # Kafka□群地址（多个用逗号隔开）
    client-id: my-app                  # 客户端标识，可选，用于区分不同客户端
    properties:                        # 传递给底层Kafka客户端的自定义参数
      security.protocol: PLAINTEXT     # 连接协议（PLAINTEXT、SASL_PLAINTEXT等）
     producer:
      acks: all                        # 确认级别：0（不等确认）、1（leader确认）、all（所有副本确认）
      retries: 3                       # 发送失败时的重试次数
      batch-size: 16384                # 批次大小（字节）
      buffer-memory: 33554432          # 生产者缓冲区内存大小
      linger-ms: 10                    # 批次发送延迟时间（毫秒）
      compression-type: gzip           # 压缩方式: none/gzip/snappy/lz4/zstd
      enable-idempotence: true         # 开启幂等性，防止重复消息
      key-serializer: org.apache.kafka.common.serialization.StringSerializer
      value-serializer:
org.springframework.kafka.support.serializer.JsonSerializer
      transaction-id-prefix: tx-           # 启用事务
      properties:                      # 额外的Kafka原生参数
        max.request.size: 1048576      # 单条消息最大大小（1MB）
```

- 当你做完上述配置，`Spring` 框架会自动
  - 创建一个 `DefaultKafkaProducerFactory`
    - 这个工具类用于加载配置并创建** `KafkaProducer` 实例**，用于和Kafka集群通信
    - `KafkaTemplate` 底层是委托 **`KafkaProducer`** 通信的
  - 创建一个 `KafkaTemplate`
  - 自动注入到你任何使用 `KafkaTemplate` 的类中
- 所以你可以直接写业务逻辑，下面是个例子
  - 当然 `KafkaTemplate` 有许多重载的方法，需要边用边学，支持同步、异步发送

```java
@Service
public class KafkaProducerService {

    private final KafkaTemplate<String, String> kafkaTemplate;

    public KafkaProducerService(KafkaTemplate<String, String> kafkaTemplate) {
        this.kafkaTemplate = kafkaTemplate;
    }

    public void send(String topic, String msg) {
        kafkaTemplate.send(topic, msg);
        System.out.println("✅ 发送成功: " + msg);
    }
}
```

- 如果想自定义功能（如拦截器、事务），可以自己手动创建 `KafkaTemplate` Bean
  - 在配置类里声明，并让启动类可以扫描到
  - 需要使用到 `ProducerFactory` 为生产者创建模板，也就是注入我们提供的配置

```java
@Configuration
public class KafkaProducerConfig {

    @Bean
    public KafkaTemplate<String, String> kafkaTemplate(
            ProducerFactory<String, String> producerFactory) {
```

```
        return new KafkaTemplate<>(producerFactory);
    }
}
```

两种发送消息的格式

- 常规发送

```
// send(String topic, Integer partition, K key, V value)

kafkaTemplate.send("test-topic", 0, "myKey", "Hello Partition 0!");
```

- 可附加元信息
    - 使用 `MessageBuilder` 配置消息
        - `withPayload`：配置消息内容
        - `KafkaHeaders.TOPIC`：消息要发送的主题
        - `KafkaHeaders.PARTITION`：消息要发送的分区
        - `KafkaHeaders.KEY`：消息的Key
        - `KafkaHeaders.TIMESTAMP`：消息发送时的时间戳

```
kafkaTemplate.send(
    MessageBuilder.withPayload("Hello Partition & Header")
        .setHeader(KafkaHeaders.TOPIC, "test-topic")
        .setHeader(KafkaHeaders.PARTITION_ID, 1)    // 指定分区
        .setHeader(KafkaHeaders.MESSAGE_KEY, "myKey") // 指定 key
        .setHeader("traceId", "trace-001")          // 自定义 header
        .build()
);
```

异步发送

`SpringKafka` 默认使用异步发送

- 返回类型为 `CompletableFuture`，我们需要使用它的 `whenComplete` 方法获取异步结果

```
kafkaTemplate.send("topic", "key", "message")
    .whenComplete((result, ex) -> {
        if (ex != null) {
            System.err.println("发送失败: " + ex.getMessage());
        } else {
            System.out.println("发送成功, offset=" +
result.getRecordMetadata().offset());
        }
    });
```

请求-响应模式

使用 `ReplyingKafkaTemplate` 实现，它支持 同步或异步等待响应消息，这使 Kafka 也能像 RPC那样工作

- 首先进行配置类的声明，需要在这声明消费者工厂，它创建的消费者实例（监听消费者）会一直监听业务消费者返回值到达的那个**topic**

    - Spring并不会帮我们自动创建 `ReplyingKafkaTemplate`

```
@Configuration
public class KafkaReplyConfig {

    // 生产者工厂
    @Bean
    public ProducerFactory<String, String> producerFactory() {
        Map<String, Object> props = new HashMap<>();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
        return new DefaultKafkaProducerFactory<>(props);
    }

    // 消费者工厂（用于回复容器）
    @Bean
    public ConsumerFactory<String, String> consumerFactory() {
        Map<String, Object> props = new HashMap<>();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "reply-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
        return new DefaultKafkaConsumerFactory<>(props);
    }

    // 监听 "reply-topic" 的容器（ReplyingKafkaTemplate 需要）
    @Bean
    public KafkaMessageListenerContainer<String, String> replyContainer(
            ConsumerFactory<String, String> cf) {

        ContainerProperties containerProperties = new ContainerProperties("reply-topic");
        return new KafkaMessageListenerContainer<>(cf, containerProperties);
    }

    // ReplyingKafkaTemplate —— 核心组件
    @Bean
    public ReplyingKafkaTemplate<String, String, String> replyingKafkaTemplate(
            ProducerFactory<String, String> pf,
            KafkaMessageListenerContainer<String, String> replyContainer) {
        return new ReplyingKafkaTemplate<>(pf, replyContainer);
    }
}
```

- 进行依赖注入并发送消息

```java
@Service
public class RequestProducer {

    @Autowired
    private ReplyingKafkaTemplate<String, String, String> replyingKafkaTemplate;

    public String sendAndReceive(String data) throws Exception {
        // 构造请求消息
        ProducerRecord<String, String> record =
                new ProducerRecord<>("request-topic", data);

        // 指定响应主题（Reply-To）
        record.headers().add(new RecordHeader(
                KafkaHeaders.REPLY_TOPIC, "reply-topic".getBytes()));

        // 发送并等待响应（同步）
        RequestReplyFuture<String, String, String> future =
                replyingKafkaTemplate.sendAndReceive(record);

        // 等待结果（阻塞直到 reply 到达）
        ConsumerRecord<String, String> response = future.get(10, TimeUnit.SECONDS);

        System.out.println("收到响应: " + response.value());
        return response.value();
    }
}
```

消费者

调用流程

```
@KafkaListener
    ↓ (指定或默认)
ContainerFactory (工厂)
    ↓ (创建和管理)
MessageListenerContainer (容器)
    ↓ (管理消费线程)
KafkaConsumer (实际消费者)
    ↓ (拉取消息)
Kafka Broker
```

使用步骤

- 引入依赖

```xml
<dependency>
  <groupId>org.springframework.kafka</groupId>
  <artifactId>spring-kafka</artifactId>
  <version>3.3.10</version>
</dependency>
```

- 进行消费者配置

```
spring:
  kafka:
    bootstrap-servers: localhost:9092  # Kafka集群地址（多个用逗号隔开）
    client-id: my-app                  # 客户端标识，可选，用于区分不同客户端
    properties:                        # 传递给底层Kafka客户端的自定义参数
      security.protocol: PLAINTEXT     # 连接协议（PLAINTEXT、SASL_PLAINTEXT等）
    consumer:
      group-id: my-group               # 消费者组ID
      auto-offset-reset: earliest      # 无偏移量时从最早消息开始消费
      enable-auto-commit: true         # 是否自动提交offset
      auto-commit-interval: 1000       # 自动提交间隔（毫秒）
      max-poll-records: 500            # 每次拉取的最大消息数量
      fetch-min-size: 1                # 每次拉取的最小字节数
      fetch-max-wait: 500              # 服务器等待数据返回的最大时间(ms)
      key-deserializer: org.apache.kafka.common.serialization.StringDeserializer
      value-deserializer: 
org.springframework.kafka.support.serializer.JsonDeserializer
      properties:
        spring.json.trusted.packages: "*"   # 允许反序列化任意包的对象
```

- 当你做完上述配置后，Spring会自动：
  - 自动创建 **`ConsumerFactory`**
  - 自动创建 **`ConcurrentKafkaListenerContainerFactory`**
  - 自动注入这些 Bean 到 `@KafkaListener` 所用的监听容器
- 使用 `@KafkaListener` 注解进行消费端的声明
  - 默认的 `factory` 会读取 **`spring.kafka.consumer.\*`** 配置

```
@KafkaListener(topics = "test-topic")
public void listen(String message) {
    System.out.println("收到消息： " + message);
}
```

- 消费带 `Header` 的消息
  - 用 `@Header` 可以拿到消息头的各种信息，例如：
    - `KafkaHeaders.RECEIVED_TOPIC`
    - `KafkaHeaders.RECEIVED_PARTITION`
    - `KafkaHeaders.OFFSET`
    - `KafkaHeaders.RECEIVED_TIMESTAMP`
    - 自定义 header（比如 traceId）

```
@KafkaListener(topics = "orders")
public void listenWithKey(@Header(KafkaHeaders.RECEIVED_KEY) String key,
                          @Payload String value) {
    System.out.println("key=" + key + ", value=" + value);
}
```

自定义容器工厂

用于自定义 `ContainerFactory`

- 控制并发消费
- 消息批量处理
- 等等

步骤

- 首先进行工厂的配置

```
@Bean
public ConcurrentKafkaListenerContainerFactory<String, String>
kafkaListenerContainerFactory() {
    ConcurrentKafkaListenerContainerFactory<String, String> factory =
            new ConcurrentKafkaListenerContainerFactory<>();
    factory.setConsumerFactory(consumerFactory()); // 自定义 ConsumerFactory
    factory.setConcurrency(3); // 并发线程数
    factory.setBatchListener(true); // 批量消费
    return factory;
}
```

- 然后在 `@KakfaListener` 使用

```
@KafkaListener(topics = "test-topic", containerFactory =
"kafkaListenerContainerFactory")
public void listen(String message) { ... }
```

手动提交偏移量

Kafka默认是自动提交偏移量到 `__consumer_offsets` 的，想要实现手动提交 `offset` ，需要进行如下工作

- 关闭自动提交

```
spring:
 kafka:
  consumer:
      enable-auto-commit: false # 关闭自动提交
```

- 创建手动提交的 `ContainerFactory`
    - 常见的 `AckMode` ：
        - `RECORD` ：每处理完一条消息自动提交
        - `BATCH` ：每批消息处理完自动提交
        - `TIME` ：定时提交
        - `COUNT` ：每处理固定数量的消息提交
        - `MANUAL` ：需要显式调用 `ack.acknowledge()`
        - `MANUAL_IMMEDIATE` ：立即提交offset

```
@Configuration
public class KafkaManualAckConfig {

    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, String> manualFactory(
            ConsumerFactory<String, String> consumerFactory) {

        ConcurrentKafkaListenerContainerFactory<String, String> factory =
                new ConcurrentKafkaListenerContainerFactory<>();
        factory.setConsumerFactory(consumerFactory);

        // 设置手动确认模式

factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.MANUAL_IMMEDIAT

        return factory;
    }
}
```

- 在 `@KafkaListener` 使用 `Acknowledgement` 接口实现自动提交

```
 @KafkaListener(topics = "manual-topic", containerFactory = "manualFactory")
    public void consume(ConsumerRecord<String, String> record, Acknowledgment ack) {
        try {
            System.out.println("收到消息: " + record.value());
            // 手动提交
            ack.acknowledge();
            System.out.println("消息已确认并提交 offset");

        } catch (Exception e) {

            System.err.println("处理消息失败: " + record.value());
        }
    }
```

请求-响应模式

上文在生产者端已经提到，在这只讲如何进行配置

- 在 `@KafkaListener` 注解的方法上加上 `@SendTo` 注解，里面填入返回值到达的Topic

```
@Component
public class ReplyConsumer {

    // 监听请求主题，并自动将返回值发送到 reply-topic
    @KafkaListener(topics = "request-topic", groupId = "reply-group")
    @SendTo("reply-topic")  // 表示自动回复到这个主题
    public String handleRequest(String message) {
        System.out.println("收到请求: " + message);
        return "处理完成: " + message.toUpperCase();
    }
}
```

异常处理与重试

当Kafka消费者出现异常时，可以实现自动重试，将失败消息发送到死信队列等功能

- 关闭自动提交offset

```
spring:
 kafka:
  consumer:
      enable-auto-commit: false # 关闭自动提交
```

- 注册 `DefaultErrorHandler`，注意是配置类
  - 使用 **`DeadLetterPublishingRecoverer`** 配置死信队列
    - 传入 `KafkaTemplate`，原始失败消息和导致失败的异常
  - 使用 **`FixedBackOff`** 配置重试间隔和次数
  - 使用 **`DefaultErrorHandler`** 创建错误处理器
    - 传入 `DeadLetterPublishingRecoverer` 和 `FixedBackOff`

```
@Configuration
public class KafkaConsumerConfig {

    @Bean
    public DefaultErrorHandler errorHandler(KafkaTemplate<Object, Object> template)
{

        // 配置 DLT（死信队列）转发逻辑
        DeadLetterPublishingRecoverer recoverer = new DeadLetterPublishingRecoverer(
            template,
            (record, ex) -> new TopicPartition("main-topic-dlt", record.partition())
        );

        // 设置重试间隔和次数（每次间隔 2s，最多重试 3 次）
        FixedBackOff backOff = new FixedBackOff(2000L, 3);

        // 生成错误处理器
        DefaultErrorHandler errorHandler = new DefaultErrorHandler(recoverer,
backOff);

        // 配置哪些异常不重试（比如非法参数）
        errorHandler.addNotRetryableExceptions(IllegalArgumentException.class);

        // 打印日志（可选）
        errorHandler.setRetryListeners((record, ex, deliveryAttempt) ->
            System.err.printf("重试第 %d 次失败: %s%n", deliveryAttempt, ex.getMessage())
        );

        return errorHandler;
    }
}
```

- 你也可以使用 `@RetryableTopic` 注解进行异常处理和重试的配置

```
@RetryableTopic(
    attempts = "4", // 总共尝试 4 次
    backoff = @Backoff(delay = 2000), // 每次间隔 2s
    dltTopicSuffix = "-dlt" // 死信队列后缀
)
@KafkaListener(topics = "main-topic", groupId = "test-group")
public void consume(String message) {
    System.out.println("收到消息: " + message);
    throw new RuntimeException("模拟异常");
}
```

Kafka Connect

两步

- 写配置文件
- 使用curl进行配置的提交

Kafka Streams

引入依赖

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-stream-binder-kafka-streams</artifactId>
</dependency>
```

写配置文件

- `spring.cloud.stream.function.definition`：告诉 `Spring Cloud Stream` 要激活哪个函数Bean
  - 如果有多个函数bean，可以使用 `;` 连接起来
- `spring.cloud.stream.function.bindings`：用于定义逻辑函数的输入输出通道，通道对应的**Topic**，序列化、消费组、分区等
  - 命名规律：定义函数的输入输出
    - `{function-name}-in-{index}`
    - `{function-name}-out-{index}`
  - `destination`：Kafka的Topic名称
- `spring.cloud.stream.kafka.streams.binder`：
  - `application-id`：Kafka Streams应用的唯一ID
  - `brokers`：Kafka集群地址
  - `configuration`：Kafka Streams 的底层配置参数
    - `commit.interval.ms`：控制状态更新提交到 `changelog topic` 的间隔
    - `cache.max.bytes.buffering`：缓存大小（控制 RocksDB 内缓存）
    - `state.dir`：本地状态存储目录

        - `processing.guarantee`：处理语义
            - `at_least_once` ：至少一次
            - `exactly_once_v2`：恰好一次
        - `default.key.serde`：默认 key 的序列化器
        - `default.value.serde`：默认 value 的序列化器
        - `num.stream.threads`：Kafka Streams 工作线程数
- `spring.cloud.stream.binders`：一个项目连接多个Kafka集群的时候使用

```yaml
spring:
  cloud:
    stream:
      function:
        definition: process
      bindings:
        process-in-0:
          destination: input-topic
        process-out-0:
          destination: output-topic
      kafka:
        streams:
          binder:
            application-id: uppercase-app
            brokers: localhost:9092
            configuration:
              commit.interval.ms: 1000
              cache.max.bytes.buffering: 10485760
```

编写函数式Bean

- 写一个配置类，将函数式接口声明为Bean对象
- 定义一个函数式接口： `Function<KStream<String, String>, KStream<String, String>>`
- 常见KStream API

```java
@Configuration
public class KafkaStreamsConfig {

    @Bean
    public Function<KStream<String, String>, KStream<String, String>> process() {
        return input -> input
                .filter((key, value) -> value.contains("hello"))
                .mapValues(String::toUpperCase);
    }
}
```

基于磁盘的消息队列

> □计目标：□磁盘存储要像内存一样快

Kafka是基于磁盘的消息队列，它收到的消息都会存储在硬盘中，而不是像Redis那样存储在内存中

- 每个 `Topic` 会被拆分成多个 `Partition`
- 每个 `Partition` 就对应一个目录，里面存放多个数据文件
    - 日志分为多个 **Segment** 文件：
        - `.log`：消息数据
        - `.index`：偏移量 → 文件位置
        - `.timeindex`：时间戳 → 偏移量

```
/kafka-logs/
 ├── topicA-0/
 │    ├── 00000000000000000000.log
 │    ├── 00000000000000000000.index
 │    └── 00000000000000000000.timeindex
 ├── topicA-1/
 │    ├── ...
```

Kakfa为什么那么快?

- 采用顺序写入的方案，也就是只追加写，可以把速度提升至600MB/s

- `Page Cache`：操作系统还会把最近访问的磁盘页缓存到内存，加快读取效率

- 零拷贝机制：使用Linux的 **`sendfile()`** 系统调用 来实现零拷贝传输，让数据从磁盘直接到达网卡，避免用户态/内核态的切换

- Kafka在三个环节都使用批处理机制，将多条消息打包在一起发送，实现高吞吐

    - Producer 批量发送
    - Broker 批量刷磁盘
    - Consumer 批量拉取

- 通过时间戳和偏移量进行文件内容的定位，使读出和写入都做到常数级别的时间复杂度

    - `.index`：偏移量 → 物理位置
    - `.timeindex`：时间戳 → 偏移量

KRaft

Kafka自身实现的元数据管理模式，用**Raft**协议取代ZooKeeper

- **RAFT** 协议是一种分布式一致性算法，它的目标是让多个节点在分布式系统中达成数据的一致性，它把分布式一致问题分解成三个子问题：
    - 领导选举
    - 日志复制
    - 安全性
- 在KRaft中，一个节点既可以做 `Broker`，也可以做 `Controller`
    - Controller有两种类型
        - `Controller Leader` 负责管理集群的元数据和协调各种操作，如选取**Leader**，创建**Topic**，管理**Partition**等
        - `Controller Follower` 主要负责备份和容灾

实现方式

- 在Kafka Broker本身的配置文件 `server.properties` 进行配置，如果是多节点，所有结点的配置必须相同！！！
    - ** `process.roles` **：指明这个节点的角色
    - ** `node.id` **：节点的唯一标识id
    - ** `listeners` **：节点的监听端口

    - ** `PLAINTTEXT` **：普通消息端口，生产者/消费者客户端连接这个端口
    - ** `CONTROLLER` **：内部 `Controller` 通信端口，用于同步元数据
  - ** `metadata.log.dir` **：源数据存储路径
  - **`controller.quorum.voters`** ：**Raft**集群的投票节点列表，用于选举 **`Controller leader`**
  - ** `log.dirs` **：消费日志目录
  - ** `num.partitions` **：新创建的topic默认分区数
  - ** `default.replication.factor` **：新topic默认副本数

```
# -------------------------
# 节点角色
# -------------------------
process.roles=broker,controller   # 同时做 broker 和 controller
node.id=1                         # 节点唯一 ID

# -------------------------
# 监听端口
# -------------------------
listeners=PLAINTEXT://localhost:9092,CONTROLLER://localhost:9093

# -------------------------
# 元数据存储目录
# -------------------------
metadata.log.dir=/tmp/kraft-metadata-logs

# -------------------------
# Controller Raft quorum（单节点自己投自己）
# -------------------------
controller.quorum.voters=1@localhost:9093

# -------------------------
# 消息日志目录
# -------------------------
log.dirs=/tmp/kafka-logs

# -------------------------
# 默认 topic 配置
# -------------------------
num.partitions=1
default.replication.factor=1
```

选举流程

- 启动时，每个 `Broker` 加入**controller quorum**
- 通过 Raft 协议的投票机制选出 `Controller Leader`
  - 需要过半节点投票（majority vote）
- 选出的 `Controller Leader` 负责：
  - 创建/删除 topic
  - 分区副本分配
  - Leader/Follower 切换
  - ISR 更新等操作

- 其他节点作为 Follower，从 Leader 复制元数据日志
- 如果 Controller Leader 崩溃：
    - 其余节点检测超时（未收到心跳），发起新一轮投票，选出新的 Leader
    - 新 Leader 从已提交的元数据日志恢复状态

消息

消息的组成部分

- 消息的键
- 消息的值
- 生成消息的时间戳
- 消息偏移量：分区中消息的唯一标识符
- 如果携带了密钥，生产者会通过密钥计算要把消息投递到哪个分区
- 消息头，存储额外的元数据

`Kafka` 中，消息一旦被使用就不会被清除，消费者通过消息偏移量得知自己读到了哪条信息

消息传递语义

> ***Producer*** 端 **+** ***Broker*** 端 **+** ***Consumer*** 端的配置共同决定的

Kafka的三种消息传递语义：

- 最多一次：消息最多被处理一次，可能丢失，但不会重复
    - 生产者：通过设置 `acks = 0` 实现
    - 消费者：先提交 `offset`
- 至少一次：消息至少被处理一次，不会丢失，但可能重复
    - 生产者：通过 `acks=1` 或 `acks=all` 并启用重试实现
    - 消费者：先处理再提交 `offset`
- 恰好一次：每条消息只被处理一次，既不丢失也不重复
    - 通过幂等生产者 **+** 事务机制实现
    - Kafka给每个生产者分配一个 **`Producer ID`**，给每个分区维护一个序列号 **`sequence number`**
    - 生产者：在每条消息中加上唯一的序列号 `PID + sequence number`，写入数据、`offset`（消费者偏移量），然后进行事务的提交
        - 普通状态下，`offset` 是消费者提交的
        - 事务机制中，** `offset` 是生产者提交的**
    - `Broker`：记录每个分区上最新的序列号，若生产者因网络重试再次发送相同序号的消息，直接忽略
    - 消费者：从 `broker` 获取消息，并处理消息，但只能获取并处理已提交的信息

复制

Kafka的每个 `Topic` 会被拆成多个 `Partition`，每个分区又会在多台 `broker` 上复制（Replication）

- 每个分区有一个 **Leader**（主副本），其余是 **Follower**（从副本）
- 所有生产者写入、消费者读取都只与**Leader**通信
- Follower会从Leader拉取数据，使自己的日志保持一致
- Kafka 通过复制保证即使某些服务器宕机，也不会丢失数据（高可用性）

复制机制

- 使用**ISR**集合等待所有副本确认才会告诉Producer消息提交成功
- 资源开销低，性能高

当所有**ISR**中的副本都挂掉时，有两种策略

- 等待ISR中的副本恢复（默认）
  - 保证数据一致性，但可能长时间不可用
- 允许非同步副本直接当Leader（不干净选举）
  - 提高可用性，但可能丢数据

同步副本（**ISR**）

Kafka使用一个集合叫做**ISR**（同步副本集），用来存储保持同步的副本

只有在以下条件下，副本才被认为是同步的：

- Broker与控制器保持心跳
- 如果是Follower，它的日志追的够快，没有落后太多
  - **Follower**落后太多时会被踢出**ISR**，当超过某个时间未同步即刻踢出

当Leader挂掉时，新的**Leader**必须从**ISR**中选出

- 集群中有一个特殊节点 `Controller` ，负责整个分区副本分配和**Leader**选举
- `Controller` 检测到 `Leader` 宕机
- 从ISR中挑选第一个副本作为新 `Leader` ，保证数据一致
- 广播新 `Leader` 信息给其他Broker和 `Producer/Consumer`

生产者

负载均衡：生产者可以知道每个 `broker` 有哪些分区，从而通过负载均衡算法把消息发送给不同的分区

- 随机分区：自动随即分配消息到不同分区
- 按键分区：根据消息的key做哈希运算，把相同的key放到一个分区

批处理机制

- 生产者会把要发送的消息暂存在内存缓冲区中
  - 当消息大小到达 `batch.size`
  - 或者等了 `linger.ms`
  - 就会把消息发送出去
- 这样设计有利于减少I/O请求次数，提升吞吐量，但也会引入额外延迟

生产者有三种确认机制

- **Acks 0**：消息发送时，生产者不关心你的消息是否发送到了Kafka集群
  - 延迟会很低，但是加剧了数据丢失的风险
- **Acks 1**：发送消息并等待 `Leader` 确认已收到消息
  - 延迟不会很高，也保证了数据安全
- **Acks -1**：发送消息并等待** `Leader` 和所有 `follower` **确认已收到消息
  - 是三种确认机制中最慢的，但是可以保证消息写入多个磁盘，安全性最高

分区分配

Producer分区策略

- **Range**：按 key 的哈希或索引顺序均匀分布

- **RoundRobin**: 轮询分配分区
- **Sticky**（新协议常用）：一段时间内持续向同一个分区发送，直到 batch 满或超时，再切换下一个分区

消费者

拉取模式

- 消费者主动向 `broker` 请求数据
- 可以一次性拉取多条消息
- 消费者落后时可以追赶消息
- 有利于让消费者自己控制节奏和批量

推送模式

- `broker` 主动推送消息给消费者
- 消费者无法控制拉取速率

消费者组

- 每个分区同一时间只能被组内一个消费者消费
- 一个消费者可以消费不同的分区
- 消费者组内每个消费者平摊不同分区
- 使用 `group.id` 标识用一个组

消费者偏移量

- 是一个整数，表示消费者下一条要读取的消息位置
- Kafka将偏移量存储在内部 `Topic` 的 `__consumer_offsets` 中，当消费者崩溃或重启，可以从上次提交的偏移量继续消费

image-20251026155003645

在上图中，标记了4个定位：

- 消费者在处理第6条消息
- 上次提交偏移量为1，下次可以从这里恢复
- 高水位线，所有副本已经同步至第10条消息
- 日志中最新写入的消息偏移量为14
- 且消费者只能看到并读取**$\le$**高水位线的信息，避免信息丢失

紧凑压缩（**Compact**）

- Kafka会为每个消费者组维护一个特殊的Topic：`__consumer_offsets` ，用来记录每个消费者组在每个分区的offset
    - `key = consumer group + partition`
    - `value = 最新的offset`
- Kafka对 `__consumer_offsets` 启用 紧凑压缩
    - 对于相同 key（同一个 `consumer group + partition` ）只保留最新的一条记录
    - 删除过期或重复的offset

分区分配

Consumer分区分配策略

- Kafka通过分区分配策略把分区分配给组内消费者，如下
    - **Range**: 连续分区按消费者顺序分配

  - **RoundRobin**：轮询分配，平均分配分区
  - **Sticky**（新协议常用）：尽量保持上次分配不变，减少分配变动

image-20251026163046158

重平衡（**Rebalance**）

当消费者组里的消费者数发生变化，就会触发重平衡，使用分区分配策略进行动态分配分区

- 经典协议
  - 步骤：
    - 所有消费者暂停消费并提交当前 `offset`
    - 消费者发送 `JoinGroup` 请求给组协调器 `Group Coordinator`
    - 组协调器选出一个组长
    - 组长进行动态分区并上报新的分配方案
    - 组协调器通知所有消费者新的分配
    - 消费者从上次提交的 `offset` 继续消费
  - 缺点：
    - 重平衡时，全员暂停消费
    - 每次都要重新计算所有分区
    - 延迟高
- 增量协作重平衡（新版本协议）
  - 步骤
    - 新消费者发送 `JoinGroup` 给协调器
    - 协调器计算新的目标分配
    - 通知现有消费者：哪些分区需要释放
    - 被要求让渡的消费者停止消费那些分区
    - 其他未受影响的消费者照常消费
  - 未受影响的消费者无停顿，消费几乎不中止

Kafka Connect

Kafka Connect是Kafka的一个重要组件，是一种数据管道框架，用于在Kafka与外部系统之间（如 `Elasticsearch` 、`Hadoop` 等）传输数据，可实现自动化、可扩展的数据导入导出

它具有以下优势：

- 以数据为中心的管道
- 具有灵活性和可扩展性
- 具有可重用性

□念

- ** `Connectors` **：连接器，负责协调数据流
- ** `Tasks` **： `Connectors` 的执行单元，实际实现数据传输逻辑
- ** `Workers` **：运行 `Connector` 和 `Task` 的进程
- ** `Conveters` **：消息转换器，用于消息格式的转换，处理数据的序列化/反序列化
- ** `Transforms` **：更改每条消息的简单逻辑，用于在消息进入或离开 `Connector` 时修改它
- ** `Dead Letter Queue` **：死信队列，处理 `Connector` 错误消息的机制

连接器（**Connector**）

类别

- ** `Source Connector` **: 从外部系统读数据写入Kafka
- ** `Sink Connector` **: 从Kafka读数据写入外部系统

任务（**Task**）

`Task` 是 `Connector` 的执行单元，实际负责将数据从源系统复制到Kafka，或从Kafka写入目标系统

作用与特点

- 支持并行执行，提高吞吐量
- 保存偏移量和状态在 Kafka 的特定主题（Topic）中，而不是 `Task` 本身
- 一个 `Connector` 可以有多个 `Task`
- `Task` 可以随时启动、停止、重新分配
- `Task` 失败不会自动触发重新平衡，需手动或通过 REST API 重启

模式

- **`Standalone`**（单机模式）：一个 `Worker` 进程管理所有 `Connector/Task`
- **`Distributed`**（分布式模式）：多个 `Worker` 协同工作，实现负载均衡和容错

进程（**Worker**）

`Worker` 是执行 `Connector` 和 `Task` 的进程

有两种不同的工作模式

- 独立模式（**Standalone**）：单进程执行所有任务
- 分布式模式（**Distributed**）：多个Worker协作，支持自动任务分配、负载均衡和故障恢复

转换器（**Converter**）

用于消息格式的转换，处理数据的序列化/反序列化

- 将系统的原始数据转换成 `Kafka` 能处理的格式
- 从 `Kafka` 读取数据时，也将 `Kafka` 数据转换回系统可用格式
- 支持链式使用（一个 Connector 可以串联多个 Transform）

逻辑转换器（**Transform**）

和转换器不同，逻辑转换器用于

- 改变消息格式
- 添加字段、删除字段
- 改变 `Topic` 等

但其只能只处理单条消息，轻量级

死信队列（**Dead Letter Queue**）

当 `Task` 处理消息失败时，不丢弃数据，而是发送到死信队列

作用

- 遇到无效或格式错误的记录时，不丢失数据，而是写入**DLQ**主题
  - **DLQ**主题（**Dead Letter Queue Topic**）是一个专门用于存储处理失败消息的特殊主题
- `errors.tolerance` 配置：
  - `none`（默认）：遇到任何错误都会导致任务失败

    - `all`：忽略错误消息，可写入DLQ

Kafka Streams

Kafka Streams是一个轻量级的`Java`库，用来对Kafka中的消息流做实时流处理

常见流程

- 从输入`Topic`拉取消息
- 更新本地状态
- 把计算结果输出到`Topic`

概念

Stream（流）

连续不断的、按时间顺序到来的数据记录序列，是一个抽象的概念

- 无界流：数据是源源不断产生的
- stream的来源就是一个Topic里面的消息

KStream

- 是Stream概念在Kafka Streams库中的具体实现
- 是一串按时间到达的事件记录（record），每条记录有`key、value、timestamp`
- `Key`非常重要
    - 数据分发：**Key**决定了数据被写入到Topic的哪个分区，从而影响并行度
    - 状态操作的基础：所有`groupByKey()`、`join()`等操作都基于Key

KTable（表）

表示一张可更新的、动态的表，每个key只保留最新值

- 对于每个key只保留一个最新的值，新值会覆盖旧值
- 分区存储：每个KTable实例只会存储一部分数据
- 状态存储：运行时，每个`KTable`实例在本地维护着一个键值存储，用来快速查询每个Key的最新值
- 变更日志主题：当`KTable`的状态发生更新（插入、更新），这个更新操作不仅会应用到本地状态存储，还会被异步地发送到一个内部的、压缩的**Kafka Topic**，即`Changelog Topic`
    - 有利于宕机后的恢复

GlobalKTable（全局表）

在每个应用程序实例上都拥有全量数据副本的特殊表

- 全量数据加载：当Kafka Streams应用程序启动时，每个实例都会从底层的`Kafka Topic`中读取全部数据来初始化自己的`GlobalKTable`
- 本地查找：当进行流表连接时，连接操作完全在本地内存中进行，使用自己的**`GlobalKTable`**副本进行查找
- 自动更新：当底层的`Kafka Topic`有新的更新时，`GlobalKTable`会进行自动同步
- 有利于避免昂贵的网络通信，但是对内存消耗大，启动速度慢

Topology（拓扑）

Kafka Streams 应用处理逻辑的**DAG**（有向无环图）

- `Kafka Streams`程序最终都会被编译成一个拓扑图，这张图描述了：
    - 数据从哪里来（**source**）
    - 经历了哪些处理步骤（**processor**）
    - 结果往哪里去（**sink**）

- 当Streams程序启动时，就会根据这张图创建线程，进行资源分配等

DSL（Domain Specific Language）

Kafka Streams 的高级**API**，让开发者用函数式语法（链式编程）快速描述流处理逻辑

自动帮我们：

- 建立 `Topology`
- 管理 `state store`
- 设置 `changelog`
- 处理序列化和反序列化

State Store（状态存储）与 changelog（状态恢复）

`Kafka Streams` 本地存储，用来保存中间状态

- 本地持久化存储（嵌入 RocksDB 或内存）用于保存中间状态
- 当 `KTable` 的状态发生更新（插入、更新），这个更新操作不仅会应用到本地状态存储，还会被异步地发送到一个内部的、压缩的**Kafka Topic**，即 `Changelog Topic`
- 当实例失败或重启时，会从 `changelog topic` 恢复本地 state

Processor API

`Kafka Streams` 的底层原生**API**，可以手工搭建拓扑，完全控制数据处理流程，允许我们定义：

- 每个节点怎么处理数据（`Processor`）
- 节点之间怎么连接（`Topology.addProcessor`）
- 使用什么状态存储（`StateStore`）

Window（窗口）

把无界流按时间切分成窗口，方便聚合

- 将无界流按时间片段分组做聚合
- 常见窗口类型与直观含义：
    - **Tumbling**（滚动窗口）：固定大小、不重叠（例如每 1 分钟一个桶）
    - **Hopping**（滑动窗口）：固定长度、按步长滑动，可重叠（例如窗口长度 1 分钟，每 30s 滑动一次）
    - **Session**（会话窗口）：基于事件间隔，若间隔超过 `gap` 则切新会话
- 窗口聚合需要为每个窗口维护 `state`（`WindowStore`）
- 需要设置保留时间（`retention`），过期窗口会被清理
- 时间处理可基于 `event-time`（事件时间）或 `processing-time`（处理时间），常用 `event-time + watermark` 处理乱序

Watermark和GracePeriod

Watermark（水位线）

- 是一个时间进度指示器，它帮助系统确定何时关闭窗口并进行聚合计算

- 当Watermark超过某个窗口的结束时间时，系统就会开始计算窗口中的数据

- **Watermark**通过事件时间进行推进：它知道目前处理进度 达到了哪个事件时间

- **Watermark**标记的是事件时间的最大值，即已知最晚的事件时间

GracePeriod（宽限期）

- 基于事件时间来处理，判断流的事件时间是否超过当前水位线加上宽限期，是的话则进行舍弃，不是的话则纳入当前窗口进行计算

- 允许延迟到达的事件在窗口结束后的一段时间内仍然能够进入当前窗口进行聚合计算

处理语义（Processing Guarantees）

有三种模式：

- 至多一次：消息最多处理一次，可能会丢，但不会重复
- 至少一次：消息至少会被处理一次，不会丢，但可能重复
- 恰好一次：消息只会处理一次，不丢也不重复

恰好一次

目标：输入消费、状态更新、输出写入三者原子化

Kafka Streams把这三个操作放到一个事务里面，实现原子性。当真正commit之后，操作和状态才会真正更新

时间语义（Time Semantics）

Kafka Streams 支持三种时间语义：

- `Event Time`：事件在真实世界中发生的时间
- `Processing Time`：事件被 Kafka Streams 应用处理的时间
- `Ingestion Time`：事件到达 Kafka broker 时的时间