

Kafka □□□□

Spring Kafka

Kafka □□□□

1

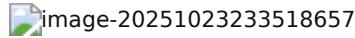
Kafka

Apache Kafka / Apache Flink

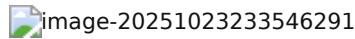
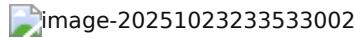
http://www.rabbitmq.com/tutorials/tutorial-one-dotnet.html RabbitMQ .NET

□□□□□□□□□□

□□□□□□□□□□□□□□□□□□/□□□□□□□□□□



A decorative horizontal bar consisting of a series of small, empty rectangular boxes arranged side-by-side.

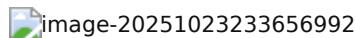


Kafka

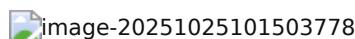
4

- Producer
 - Consumer
 - Kafka Cluster

2

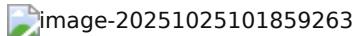


- Producer ကြောင်းပြုခြင်း Kafka broker အကြောင်း
 - Consumer ကြောင်းပြုခြင်း Kafka broker အကြောင်း
 - Consumer Group(CG) ကြောင်းပြုခြင်း consumer များ
 - ကြောင်းပြုခြင်းများ
 - ကြောင်းပြုခြင်းများ



- Broker \square * \square Kafka $\square\text{---}\square\text{---}\square$ broker * $\square\text{---}\square\text{---}\square$ broker $\square\text{---}\square$ broker $\square\text{---}\square$ topic

- Topic 话题 topic
- Partition 分区 Topic 话题 broker 节点 Topic 话题 Partition 分区 例句



- Replica 副本 topic 话题 leader 领导者 follower 追随者
- Leader 领导者 Partition 分区
- Follower 追随者 Partition 分区

Kafka API

Spring 框架 Kafka API

核心类

- KafkaTemplate 话题生产者
- @KafkaListener 话题消费者

其他辅助类和工厂类

- ProducerFactory 话题生产者工厂
- ConsumerFactory 话题消费者工厂
- ConcurrentKafkaListenerContainerFactory 并行监听器容器工厂
- KafkaProducer 话题生产者 KafkaTemplate 话题生产者工厂
- DefaultKafkaProducerFactory 话题生产者工厂 ProducerFactory 话题生产者 KafkaProducer 话题生产者
- ContainerFactory 话题消费者工厂 MessageListenerContainer 监听器容器工厂

配置

配置文件

- 配置文件

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

- 配置文件

- yaml 配置文件

```
spring:
  kafka:
    bootstrap-servers: localhost:9092 # Kafka 服务器地址
    client-id: my-app # 客户端 ID
    properties: # Kafka 通用属性
      security.protocol: PLAINTEXT # PLAINTEXT 或 SASL_PLAINTEXT
    producer:
      acks: all # 要求所有 (0) 或 1 (leader) 或 all (所有分区)
      retries: 3 # 重试次数
```

```

batch-size: 16384          # 亂行数
buffer-memory: 33554432    # バッファメモリ
linger-ms: 10               # リンガーミリ秒
compression-type: gzip      # エンコードnone/gzip/snappy/lz4/zstd
enable-idempotence: true    # 一貫性
key-serializer: org.apache.kafka.common.serialization.StringSerializer
value-serializer:
org.springframework.kafka.support.serializer.JsonSerializer
    transaction-id-prefix: tx-          # トランザクションID
    properties:                      # Kafkaプロパティ
        max.request.size: 1048576       # リクエスト最大1MB

```

- Spring Kafka

- DefaultKafkaProducerFactory
 - KafkaProducer
 - KafkaTemplate
- KafkaTemplate
 - KafkaTemplate Bean

- ProducerFactory

- KafkaTemplate Bean

```

@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("✓ 送信: " + msg);
    }
}

```

- KafkaTemplate Bean
 - ProducerFactory

```

@Configuration
public class KafkaProducerConfig {

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

```

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

二、KafkaTemplate

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

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

- `MessageBuilder`
 - `withPayload`
 - `KafkaHeaders.TOPIC`
 - `KafkaHeaders.PARTITION_ID`
 - `KafkaHeaders.KEY`
 - `KafkaHeaders.TIMESTAMP`

```
kafkaTemplate.send(
    MessageBuilder.withPayload("Hello Partition & Header")
        .setHeader(KafkaHeaders.TOPIC, "test-topic")
        .setHeader(KafkaHeaders.PARTITION_ID, 1)      // 分区
        .setHeader(KafkaHeaders.MESSAGE_KEY, "myKey") // 主键
        .setHeader("traceId", "trace-001")           // 头部
        .build()
);
```

三、Spring Kafka

- `CompletableFuture` `whenComplete`

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

四、Replying Kafka

• `ReplyingKafkaTemplate` `ReplyingKafkaTemplate` Kafka RPC

- `sendAndReceive` `sendAndReceive` **topic**

- Spring 亂子类 ReplyingKafkaTemplate

```

@Configuration
public class KafkaReplyConfig {

    // 亂子类
    @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);
    }

    // 亂子类
    @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);
    }

    // 亂子类 "reply-topic" 亂子类 ReplyingKafkaTemplate 亂子类
    @Bean
    public KafkaMessageListenerContainer<String, String> replyContainer(
        ConsumerFactory<String, String> cf) {

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

    // ReplyingKafkaTemplate — 亂子类
    @Bean
    public ReplyingKafkaTemplate<String, String, String> replyingKafkaTemplate(
        ProducerFactory<String, String> pf,
        KafkaMessageListenerContainer<String, String> replyContainer) {
        return new ReplyingKafkaTemplate<>(pf, replyContainer);
    }
}

```

- 亂子类

```

@Service
public class RequestProducer {

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

    public String sendAndReceive(String data) throws Exception {
        // 送信
        ProducerRecord<String, String> record =
            new ProducerRecord<>("request-topic", data);

        // レイプヘッダ Reply-To
        record.headers().add(new RecordHeader(
            KafkaHeaders.REPLY_TOPIC, "reply-topic".getBytes()));

        // 受信
        RequestReplyFuture<String, String, String> future =
            replyingKafkaTemplate.sendAndReceive(record);

        // レイプヘッダ reply 有り
        ConsumerRecord<String, String> response = future.get(10, TimeUnit.SECONDS);

        System.out.println("[" + response.value());
        return response.value();
    }
}

```

○○○

○○○○

```

@KafkaListener
    ↓ (○○○○)
ContainerFactory (○○)
    ↓ (○○○○)
MessageListenerContainer (○○)
    ↓ (○○○○○)
KafkaConsumer (○○○○○)
    ↓ (○○○○)
Kafka Broker

```

○○○○

- ○○○○

```

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

```

- Kafka配置

```
spring:
  kafka:
    bootstrap-servers: localhost:9092 # Kafka集群地址
    client-id: my-app # 客户端ID
    properties: # Kafka客户端属性
      security.protocol: PLAINTEXT # 使用PLAINTEXT或SASL_PLAINTEXT
    consumer:
      group-id: my-group # 消费者ID
      auto-offset-reset: earliest # 自动偏移量重置
      enable-auto-commit: true # 启用自动提交
      auto-commit-interval: 1000 # 自动提交间隔
      max-poll-records: 500 # 最大批次记录数
      fetch-min-size: 1 # 每次拉取的最小字节数
      fetch-max-wait: 500 # 每次拉取的最大等待时间(ms)
      key-deserializer: org.apache.kafka.common.serialization.StringDeserializer
      value-deserializer:
        org.springframework.kafka.support.serializer.JsonDeserializer
        properties:
          spring.json.trusted.packages: "*" # 受信的JSON包

```

- Spring Kafka API
 - ConsumerFactory
 - ConcurrentKafkaListenerContainerFactory
 - Bean @KafkaListener 容器
- @KafkaListener 容器
 - factory 属性 `spring.kafka.consumer.*`

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

- Header 属性
 - @Header 属性
 - KafkaHeaders.RECEIVED_TOPIC
 - KafkaHeaders.RECEIVED_PARTITION
 - KafkaHeaders.OFFSET
 - KafkaHeaders.RECEIVED_TIMESTAMP
 - header属性 traceId

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

容器工厂

ContainerFactory

- 容器工厂
- 容器
- 容器

容器

- 容器工厂容器

```
@Bean
public ConcurrentKafkaListenerContainerFactory<String, String>
kafkaListenerContainerFactory() {
    ConcurrentKafkaListenerContainerFactory<String, String> factory =
        new ConcurrentKafkaListenerContainerFactory<>();
    factory.setConsumerFactory(consumerFactory()); // 容器 ConsumerFactory
    factory.setConcurrency(3); // 容器数
    factory.setBatchListener(true); // 批量
    return factory;
}
```

- 容器 @KafkaListener 容器

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

消费者配置

KafkaConsumer __consumer_offsets 消费者配置 offset 消费偏移量

- 消费者配置

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

- 容器工厂 ContainerFactory

- 容器 AckMode 方式
 - RECORD 按记录提交
 - BATCH 按批次提交
 - TIME 按时间提交
 - COUNT 按数量提交
 - MANUAL 客户端调用 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);

        // ...

        factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.MANUAL_IMMEDIATE);

        return factory;
    }
}

```

- `@KafkaListener` による Acknowledgement の実装

```

@KafkaListener(topics = "manual-topic", containerFactory = "manualFactory")
public void consume(ConsumerRecord<String, String> record, Acknowledgment ack) {
    try {
        System.out.println("接続: " + record.value());
        // ...
        ack.acknowledge();
        System.out.println("確認 offset");
    } catch (Exception e) {
        System.err.println("エラー: " + record.value());
    }
}

```

MQ-メッセージ

MQを用いたデータのやり取り

- `@KafkaListener` による `@SendTo` による MQ Topic

```

@Component
public class ReplyConsumer {

    // リクエスト用のMQ topic reply-topic
    @KafkaListener(topics = "request-topic", groupId = "reply-group")
    @SendTo("reply-topic") // レply用のMQ topic
    public String handleRequest(String message) {
        System.out.println("接続" + message);
        return "確認" + message.toUpperCase();
    }
}

```

```
    }
}
```

Kafka
Kafka

- offset

```
spring:
kafka:
consumer:
  enable-auto-commit: false #
```

- DefaultErrorHandler
- DeadLetterPublishingRecoverer
 - KafkaTemplate
- FixedBackOff
- DefaultErrorHandler
 - DeadLetterPublishingRecoverer □ FixedBackOff

```
@Configuration
public class KafkaConsumerConfig {

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

    // DLT
    DeadLetterPublishingRecoverer recoverer = new DeadLetterPublishingRecoverer(
      template,
      (record, ex) -> new TopicPartition("main-topic-dlt", record.partition())
    );

    // 2s 3
    FixedBackOff backOff = new FixedBackOff(2000L, 3);

    // 
    DefaultErrorHandler errorHandler = new DefaultErrorHandler(recoverer,
    backOff);

    // 
    errorHandler.addNotRetryableExceptions(IllegalArgumentException.class);

    // 
    errorHandler.setRetryListeners((record, ex, deliveryAttempt) ->
      System.err.printf("%d %s%n", deliveryAttempt, ex.getMessage())
    );

    return errorHandler;
  }
}
```

```
    }
}
```

- `@RetryableTopic` @RetryableTopic

```
@RetryableTopic(
    attempts = "4", // 尝试 4 次
    backoff = @Backoff(delay = 2000), // 延时 2s
    dltTopicSuffix = "-dlt" // 异常主题后缀
)
@KafkaListener(topics = "main-topic", groupId = "test-group")
public void consume(String message) {
    System.out.println("接收到消息" + message);
    throw new RuntimeException("测试异常");
}
```

Kafka Connect

○○

- `Connector`
- `curl`命令行工具

Kafka Streams

○○○○

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

○○○○○

- `spring.cloud.stream.function.definition` 定义 Spring Cloud Stream `FunctionBean`
 - `function-bean` ; `name`
- `spring.cloud.stream.function.bindings` 定义 `Function` 和 `Topic` 的映射关系
 - `function-name`
 - `{function-name}-in-{index}`
 - `{function-name}-out-{index}`
 - `destination` Kafka Topic
- `spring.cloud.stream.kafka.streams.binder`
 - `application-id` Kafka Streams 应用 ID
 - `brokers` Kafka 地址
 - `configuration` Kafka Streams 配置
 - `commit.interval.ms` 提交间隔 ms, changelog topic
 - `cache.max.bytes.buffering` 缓冲区大小 RocksDB 使用
 - `state.dir` 状态目录

- processing.guarantee 保证
 - at_least_once
 - exactly_once_v2
 - default.key.serde 用于 key 读写
 - default.value.serde 用于 value 读写
 - num.stream.threads Kafka Streams 线程数
- spring.cloud.stream.binders 为 Kafka Stream 定义的 binder

```
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
```

Bean

- Java Bean
- Function<KStream<String, String>, KStream<String, String>>
- [KStream API](#)

```
@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);
  }
}
```

其他集成

| Redis

Kafka 与 Redis 集成，通过 Kafka Connect 将 Kafka 和 Redis 集成

- Topic はパーティション Partition
- Partition はデータ構造の構成要素
 - データ構造 Segment は
 - .log ファイル
 - .index ファイル → ポインター
 - .timeindex ファイル → ポインター

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

Kafka

- パーティションごとに複数のログファイル(600MB/s)
- Page Cache を利用するため読み込み速度が速い
- ファイル操作Linuxの **sendfile()** メソッドを用いて高速化している
- Kafkaの構成要素
 - Producer
 - Broker
 - Consumer
- ファイル構造
 - .index ファイル → ポインター
 - .timeindex ファイル → ポインター

KRaft

KafkaのリザーバーをRaftで実装したZooKeeper

- **RAFT** は分散システムで用いられる複数のノード間でのデータ同期と並列処理のためのアルゴリズム
- ノード
 - リーダー
 - フォロワー
 - ジョブ
- KRaftの構成要素 Broker はController
 - Controller
 - Controller Leader はリード機能を持つノード
 - Controller Follower はリード機能を持たないノード

まとめ

- Kafka Brokerの設定ファイル `server.properties` の主要な設定項目
 - `process.roles`
 - `node.id`
 - `listeners`

```

    • ** PLAINTEXT **  

    • ** CONTROLLER ** Controller  

    ○ ** metadata.log.dir **  

    ○ controller.quorum.voters Raft Controller leader  

    ○ ** log.dirs **  

    ○ ** num.partitions **topic  

    ○ ** default.replication.factor **topic

```

```

# -----  

# Broker  

# -----  

process.roles=broker,controller # broker + controller  

node.id=1 # ID  

# -----  

# Controller  

# -----  

listeners=PLAINTEXT://localhost:9092,CONTROLLER://localhost:9093  

# -----  

# Metadata logs  

# -----  

metadata.log.dir=/tmp/kraft-metadata-logs  

# -----  

# Controller Raft quorum  

# -----  

controller.quorum.voters=1@localhost:9093  

# -----  

# Kafka logs  

# -----  

log.dirs=/tmp/kafka-logs  

# -----  

# Topic  

# -----  

num.partitions=1  

default.replication.factor=1

```

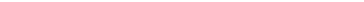
Broker

- Broker + **controller quorum**
- Raft + Controller Leader
 - majority vote
- Controller Leader
 - topic
 - ISR
 - Leader/Follower
 - ISR

- ဗိုလ်ချုပ် Follower မှာ Leader ရေးဆွဲမှု
 - မှာ Controller Leader ရေး
 - ဗိုလ်ချုပ်အသေးစိတ်အကြောင်းအရာများအတွက် Leader
 - မှာ Leader ရေးဆွဲမှု

1

6 of 7

Kafka 

6 of 6

Kafka

- **acks** **offset**
 - **acks = 0**
 - **offset**
 - **acks=1** **acks=all** **offset**
 - **offset**
 - **Producer ID** **sequence number**
 - **+ offset**
 - **Kafka** **Producer ID** **sequence number** **offset**
 - **PID + sequence number** **offset**
 - **offset** **Broker**
 - **offset** broker**
 - **Broker** **offset**
 - **broker** **offset**

1

Kafka Topic Partition broker Replication

- ဗိုလ်ချုပ် **Leader** မှာ အသေးစိတ် **Follower** များ
 - အသေးစိတ် မှာ အသေးစိတ် **Leader** မှာ
 - Follower မှာ Leader မှာ အသေးစိတ် များ
 - Kafka မှာ အသေးစိတ် များ

1

- **ISR** Producer
 -

ISR

- ISR
◦ Leader
- ISR
◦ Follower

ISR

Kafka ISR

Broker

- Broker
- Follower
- Follower** ISR

Leader Leader ISR

- Controller Leader
- Controller Leader
- ISR Leader
- Leader Broker Producer/Consumer

Broker

broker

- batch.size
- key.key

Config

- batch.size
 - linger.ms
 - linger.ms
- I/O

Acks

- Acks 0** Kafka
- Acks 1** Leader
- Acks -1** Leader follower

Producer

- Range key

- **RoundRobin** 亂抐柧
- **Sticky** 亂抐柧 僨僗僋僩僢僾 僨僗僋僩僢僾 batch 僨僗僋僩僢僾

亂抐柧

亂抐柧

- broker 亂抐柧
- 亂抐柧 僨僗僋僩僢僾
- 亂抐柧 僨僗僋僩僢僾
- 亂抐柧 僨僗僋僩僢僾

亂抐柧

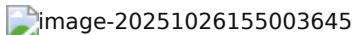
- broker 亂抐柧 僨僗僋僩僢僾
- 亂抐柧 僨僗僋僩僢僾

亂抐柧

- 亂抐柧 僨僗僋僩僢僾
- 亂抐柧 僨僗僋僩僢僾
- 亂抐柧 僨僗僋僩僢僾
- group.id 亂抐柧

亂抐柧

- 亂抐柧 僨僗僋僩僢僾
- Kafka Topic __consumer_offsets 亂抐柧 僨僗僋僩僢僾



亂抐柧4

- 亂抐柧6
- 亂抐柧1
- 亂抐柧10
- 亂抐柧14
- 亂抐柧\$\\le\$

亂抐柧Compact

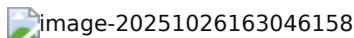
- Kafka Topic __consumer_offsets offset
 - key = consumer group + partition
 - value = offset
- Kafka __consumer_offsets offset
 - key consumer group + partition offset
 - value offset

亂抐柧

Consumer

- Kafka Topic offset
 - Range

- **RoundRobin** 亂序發送
 - **Sticky** 貼合發送



Rebalance

- **JoinGroup**
 - **JoinGroup**
 - **JoinGroupRequest** offset
 - **JoinGroupResponse** Group Coordinator
 - **JoinGroupRequest**
 - **JoinGroupResponse**
 - **JoinGroupRequest** offset 亂數
 - **LeaveGroup**
 - **LeaveGroupRequest**
 - **LeaveGroupResponse**
 - **LeaveGroupRequest**
 - **Rebalance**
 - **Rebalance**
 - **RebalanceRequest** JoinGroup 亂數
 - **RebalanceResponse**
 - **RebalanceRequest**
 - **RebalanceResponse**
 - **RebalanceRequest**
 - **PartitionReassignment**
 - **PartitionReassignmentRequest**
 - **PartitionReassignmentResponse**

Kafka Connect

Kafka Connect □ Kafka □ Elasticsearch □ Hadoop □

1

- 
 - 
 - 

11

- **Connectors** 連接器
 - **Tasks** 任务 Connector 連接器
 - **Workers** 工作者 Connector 連接器 Task 任务
 - **Converters** 转换器 連接器/转换器
 - **Transforms** 变换器 連接器 Connector 連接器
 - **Dead Letter Queue** 死信队列 Connector 連接器

Connector

1

- ** Source Connector ** Kafka
 - ** Sink Connector ** Kafka

Task

Task □ Connector □□□□□□□□□□□□□□□Kafka□□□Kafka□□□□□

5

- Kafka Topic Task
 - Kafka Topic Task
 - Connector Task
 - Task
 - Task REST API

1

Worker

Worker Connector Task

10

- **Standalone** Worker
 - **Distributed** Worker

Converter

- Kafka Connect Kafka API
 - Kafka API Kafka Connect API
 - Kafka Connect Connector Transform API

Transform

□□□□□□□□□□

-     
 -      
 - Topic

Dead Letter Queue

□ Task □□□□□□□□□□□□□□□□□

1

- **DLQ** ကို
 - **Dead Letter Queue Topic** ဖြစ်အပေါ်
 - errors.tolerance ကို
 - none ဖြစ်အပေါ်

- all 旣存のメッセージをDLQ

Kafka Streams

Kafka Streams Kafka Streams Java API Kafka Streams API

概念

- Topic 概念
- Stream
- KStream

Stream

Stream 概念

複数のメッセージを時間順に並べたもの

- 並びの要素
- stream は Topic の Topic

KStream

- Stream は Kafka Streams API
- record は key,value,timestamp
- Key 概念
 - Key は Topic
 - groupByKey() と join() は Key

KTable

複数のメッセージを key で並べる

- key は Topic
- KTable は Topic
- KTable は Key で並ぶ
- KTable は Kafka Topic と Changelog Topic
 - Changelog Topic
 - 並びの要素

GlobalKTable

複数のメッセージを key で並べる

- GlobalKTable は Kafka Streams API で Kafka Topic が GlobalKTable である
- GlobalKTable は Topic
- Kafka Topic が GlobalKTable である
- GlobalKTable は Topic

Topology

Kafka Streams は DAG 形式

- Kafka Streams は DAG 形式
 - source
 - processor
 - sink

- `Streams` `Topology`

DSL **Domain Specific Language**

Kafka Streams `API` `Topology` `Processor API` `State Store`

• `Topology`

- `Topology`
- `state store`
- `changelog`
- `Processor API`

State Store `changelog` `Processor API`

Kafka Streams `Processor API` `Topology`

- `Changelog` RocksDB `Processor`
- `KTable` `changelog` `Processor` `Topic` **Kafka Topic**
- `changelog topic` `state`

Processor API

Kafka Streams `Processor API` `Topology` `Processor`

- `Processor`
- `Topology.addProcessor`
- `StateStore`

Window `Processor API`

• `Processor API`

- `Processor API`
- `Window`
 - **Tumbling** `Processor` `1 second`
 - **Hopping** `Processor` `1 second` `30s`
 - **Session** `Processor` `gap` `100ms`
- `Processor API` `state` `WindowStore`
- `Processor API` `retention` `Processor API`
- `Processor API` `event-time` `Processor API` `processing-time` `Processor API` `event-time + watermark` `Processor API`

Watermark `GracePeriod`

Watermark `Processor API`

- `Processor API`
- `Watermark` `Processor API`
- **Watermark** `Processor API` `Processor API`
- **Watermark** `Processor API` `Processor API`

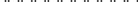
GracePeriod `Processor API`

- `Processor API`

-

Processing Guarantees

1

- 
 - 
 - 

1

Kafka Streams ━━━━━━ commit ━━━━━━

Time Semantics

Kafka Streams □□□□□□□□

- Event Time 事件时间
 - Processing Time 处理时间 Kafka Streams 处理器
 - Ingestion Time 消息摄入时间 Kafka broker 收到消息