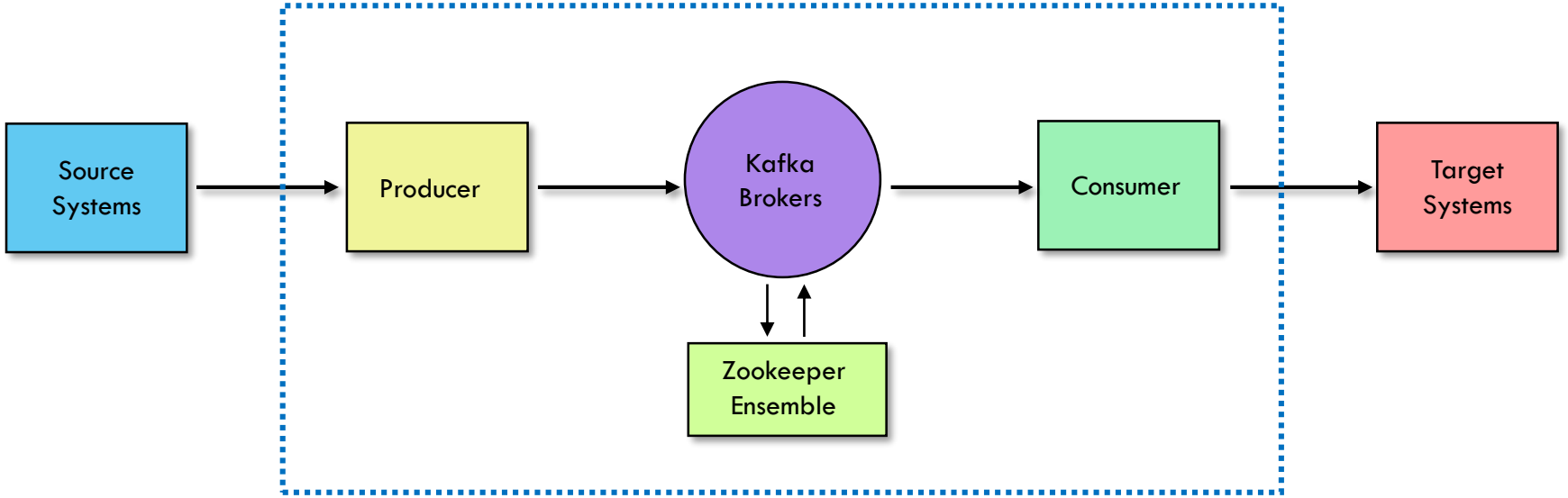
ak系列 – ak03

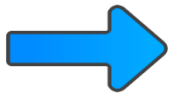
(Java Producer & Consumer – 進階)

Kafka Ecosystem:
Kafka Core

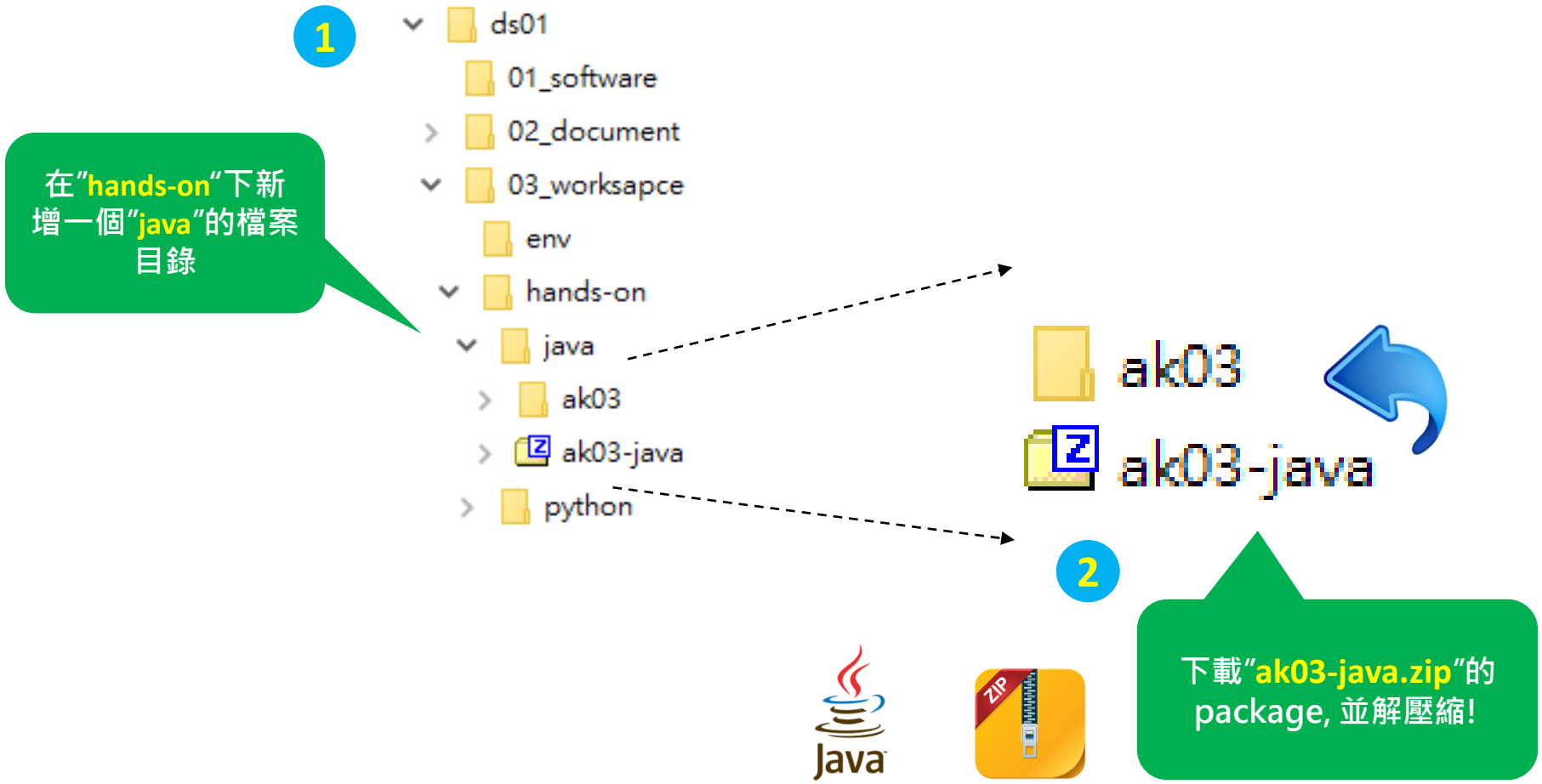


Download Sample Java Project Code

- Get Java Demo source code



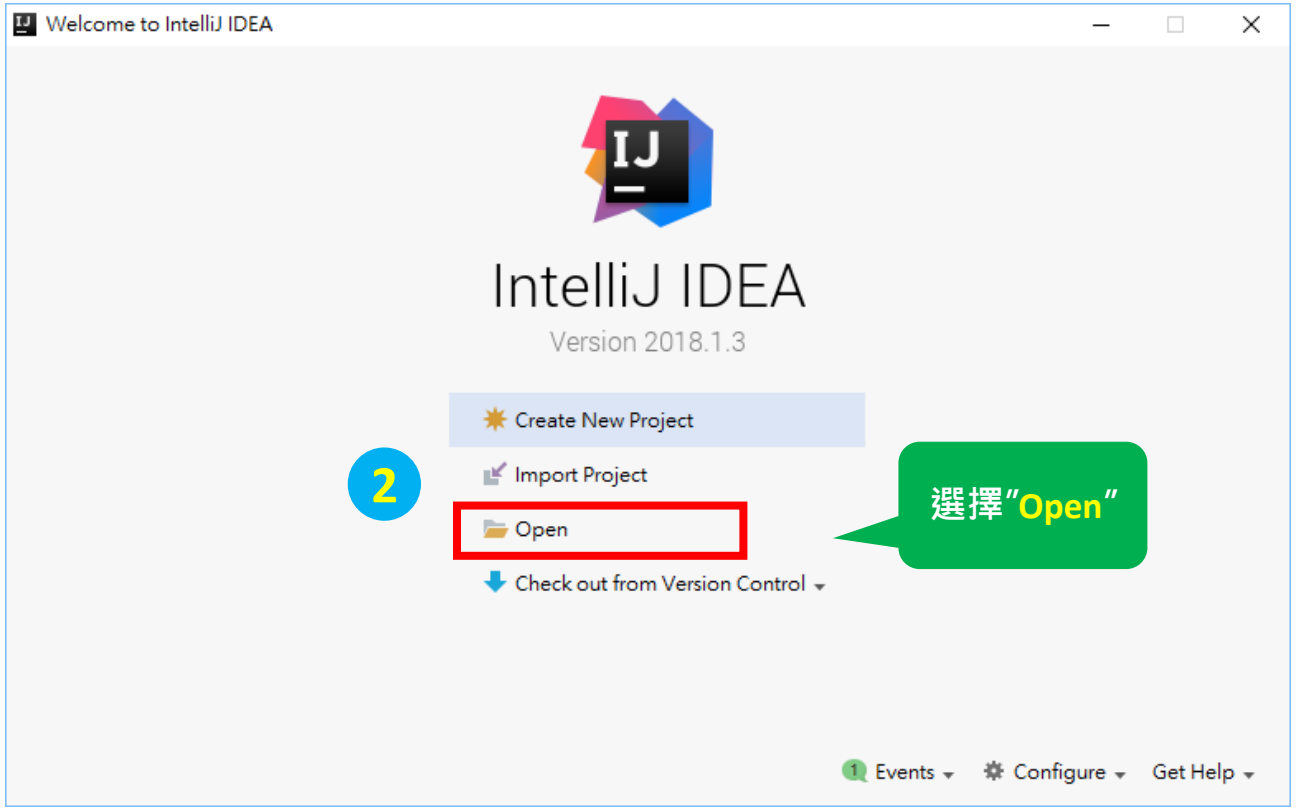
Decompress Demo Java



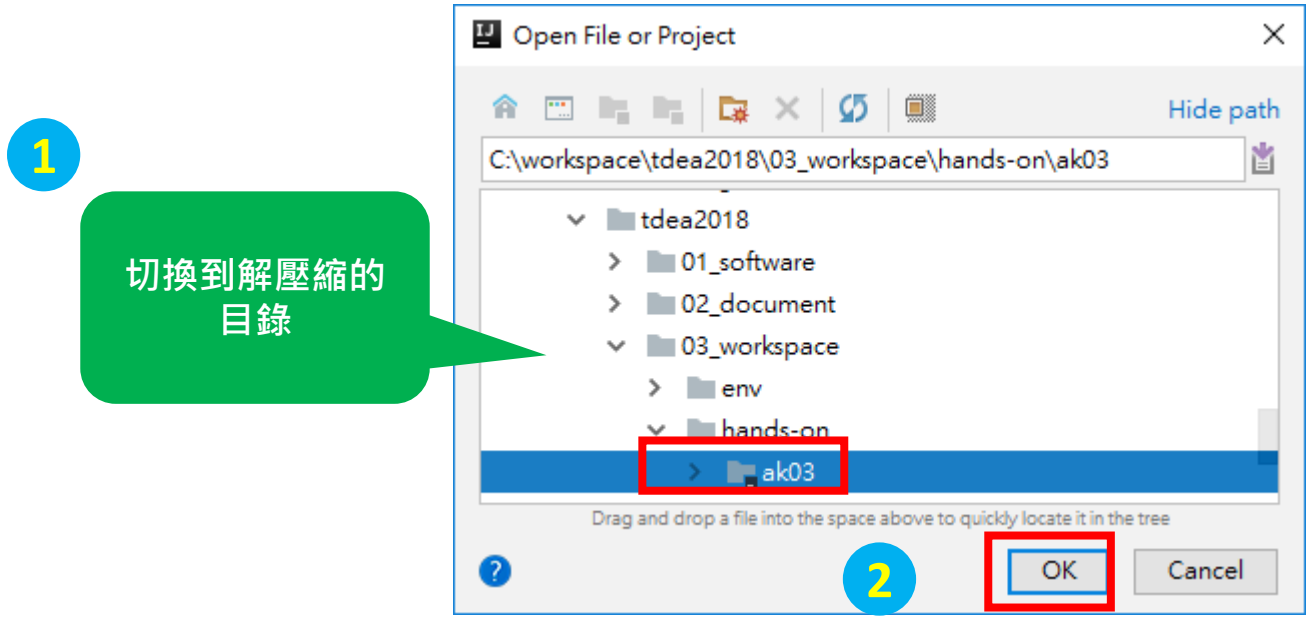
Open Demo Java Project using **IntelliJ**

1

啟動IntelliJ IDEA

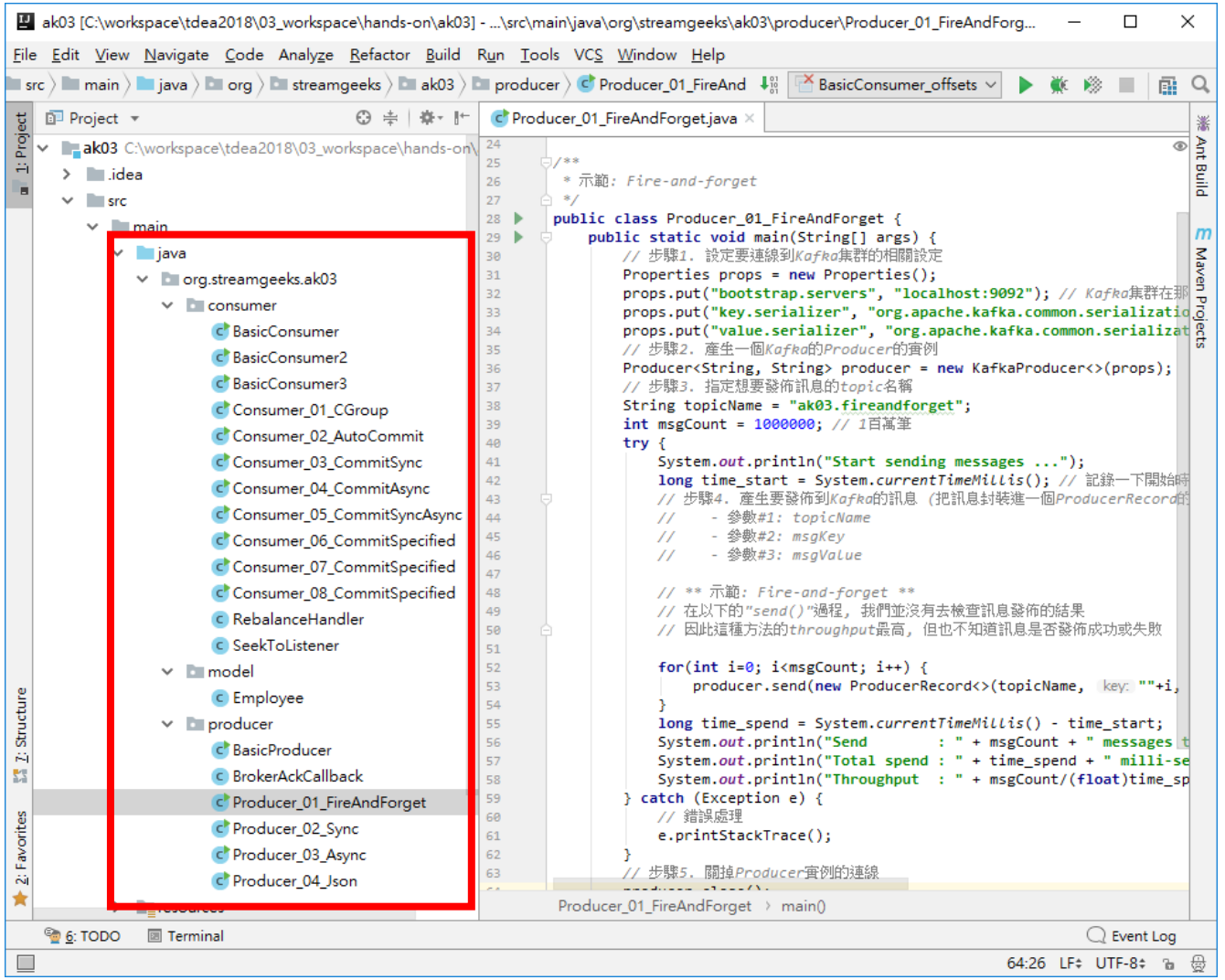


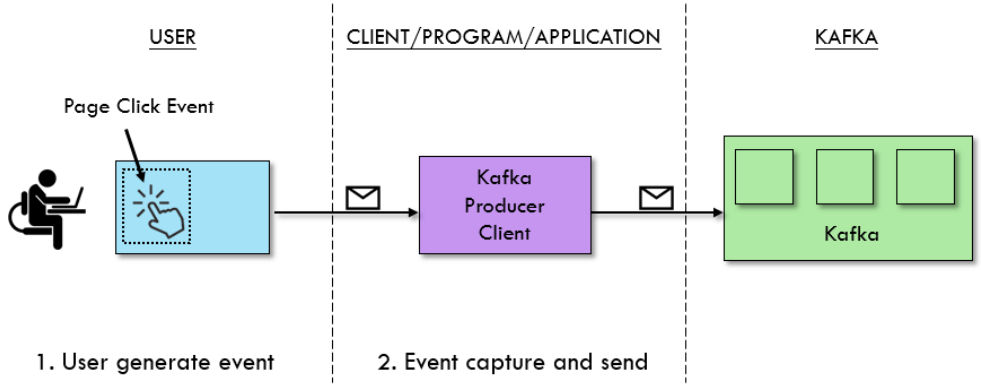
Open Demo Java Project using **IntelliJ**



Open Demo Java Project using **IntelliJ**

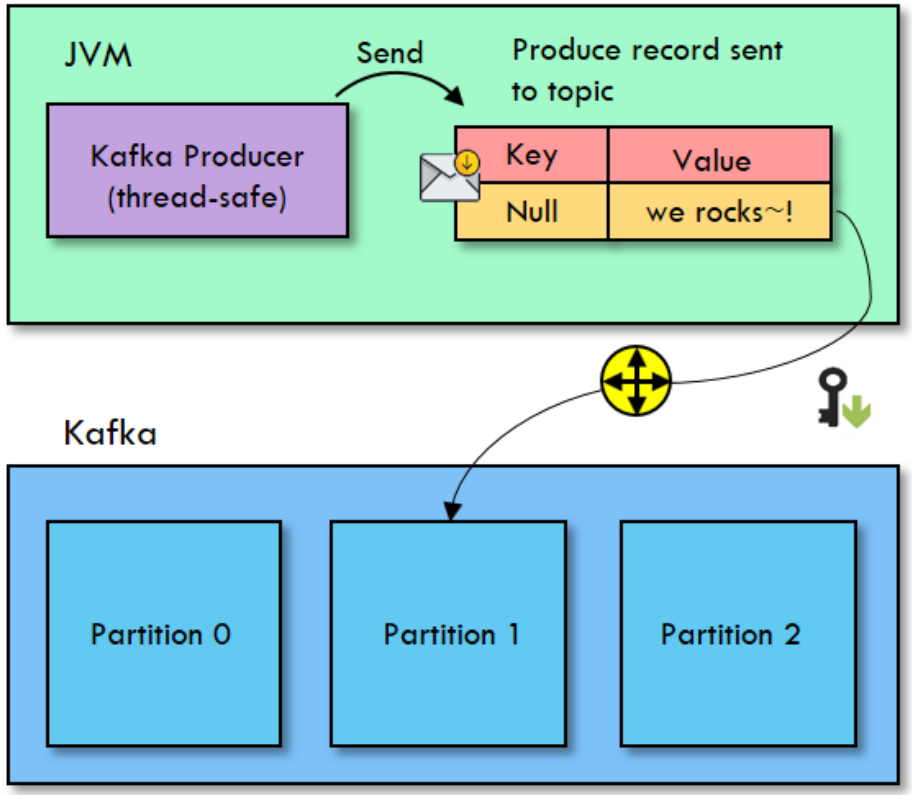
1

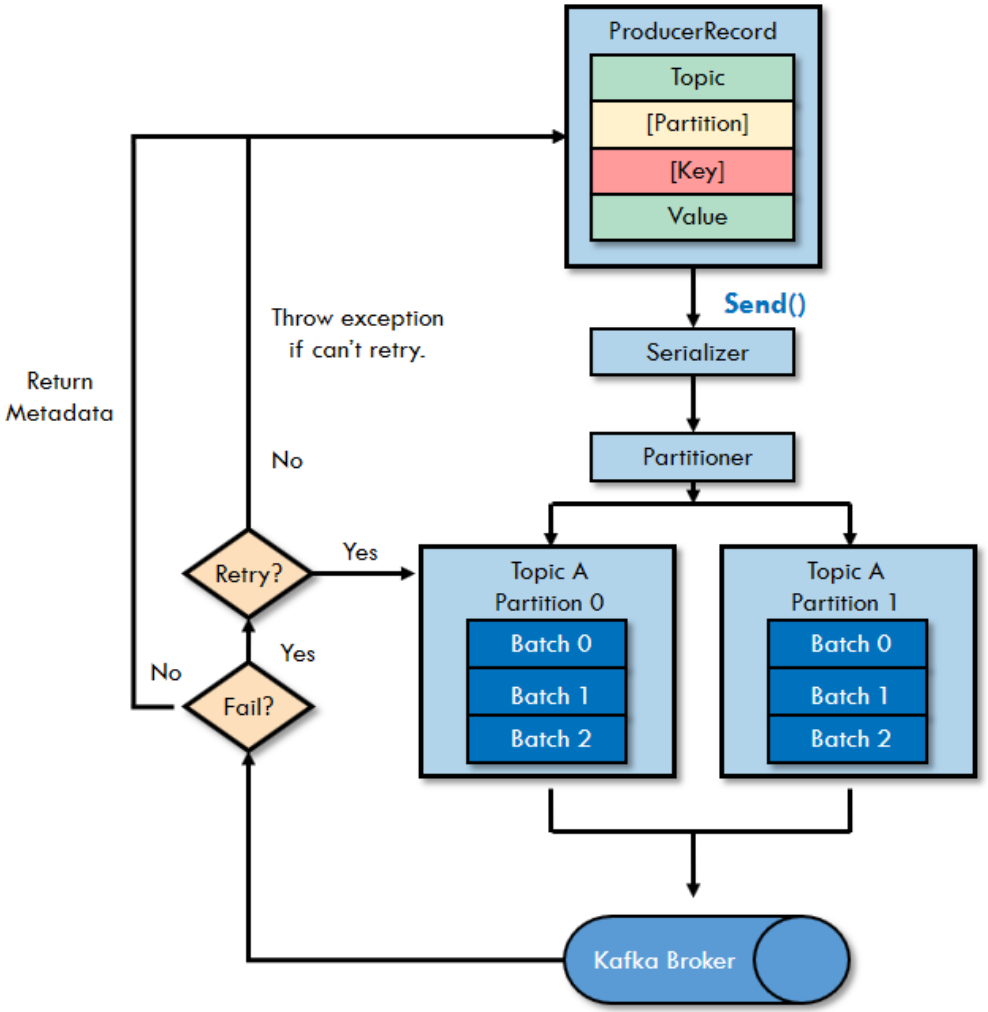




Kafka Producers: Writing Messages to Kafka

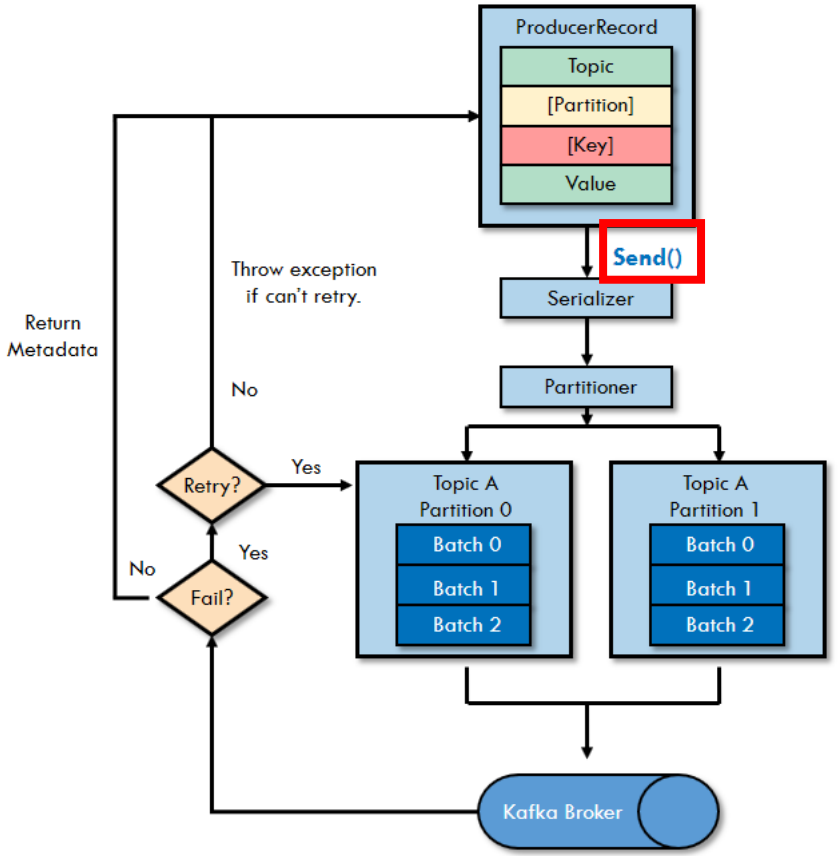
Producer Overview





Three primary methods of sending messages

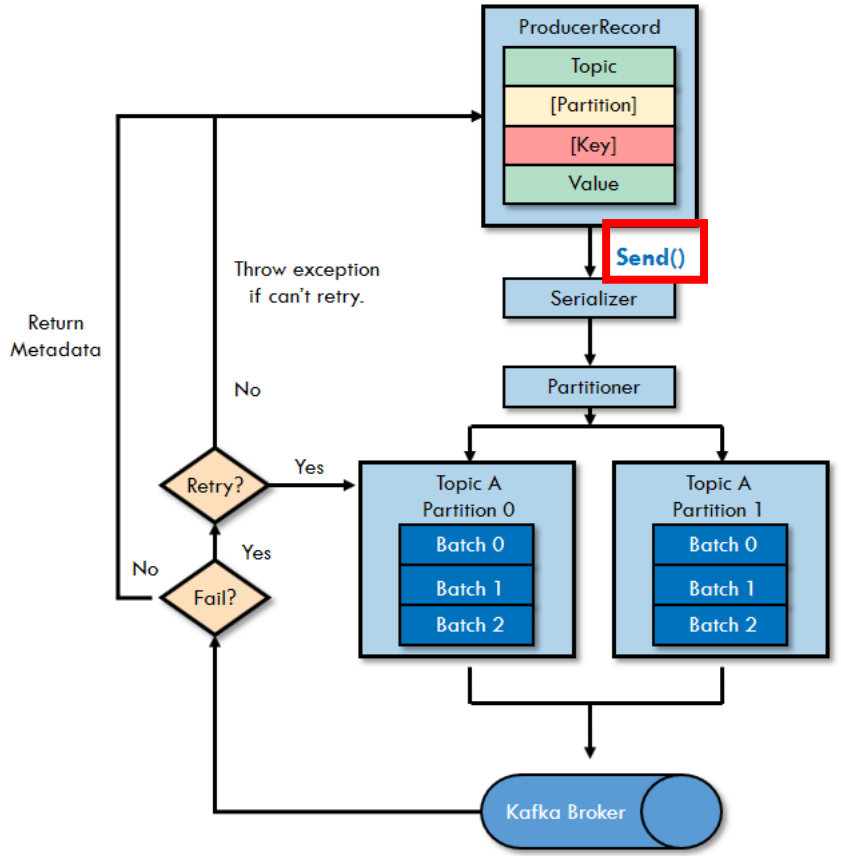
- **Fire-and-forget**
  - We send a message to the server and don't really care if it arrives successfully or not.
- **Synchronous send**
  - We send a message, and wait to see if the **send()** was successful or not.
- **Asynchronous send**
  - We call the **send()** method with a **callback function**, which gets triggered when it receives a response from the Kafka broker.



Three primary methods of sending messages
Fire-and-forget

- Producer_01_FireAndForget

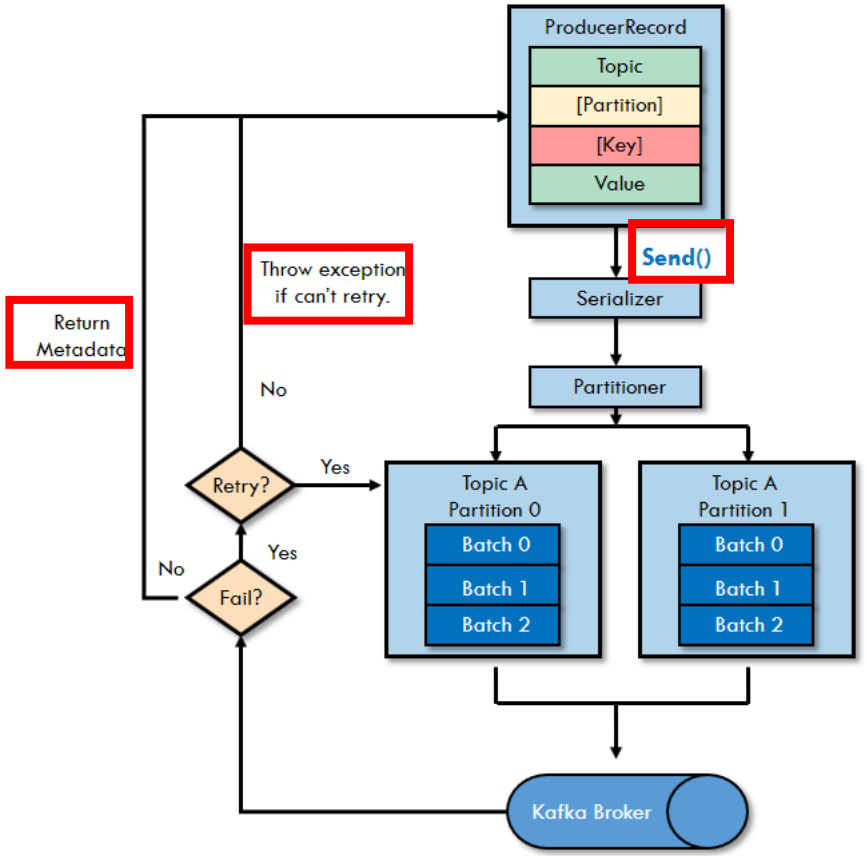
```
// ** 示範: Fire-and-forget **
// 在以下的"send()"過程, 我們並沒有去檢查訊息發佈的結果
// 因此這種方法的throughput最高, 但也不知道訊息是否發佈成功或失敗

for(int i=0; i<msgCount; i++) {
    producer.send(new ProducerRecord<>(topicName,  key: ""+i,  value: "msg_"+i));
}
```



Three primary methods of sending messages
Synchronous send

- Producer_02_Sync

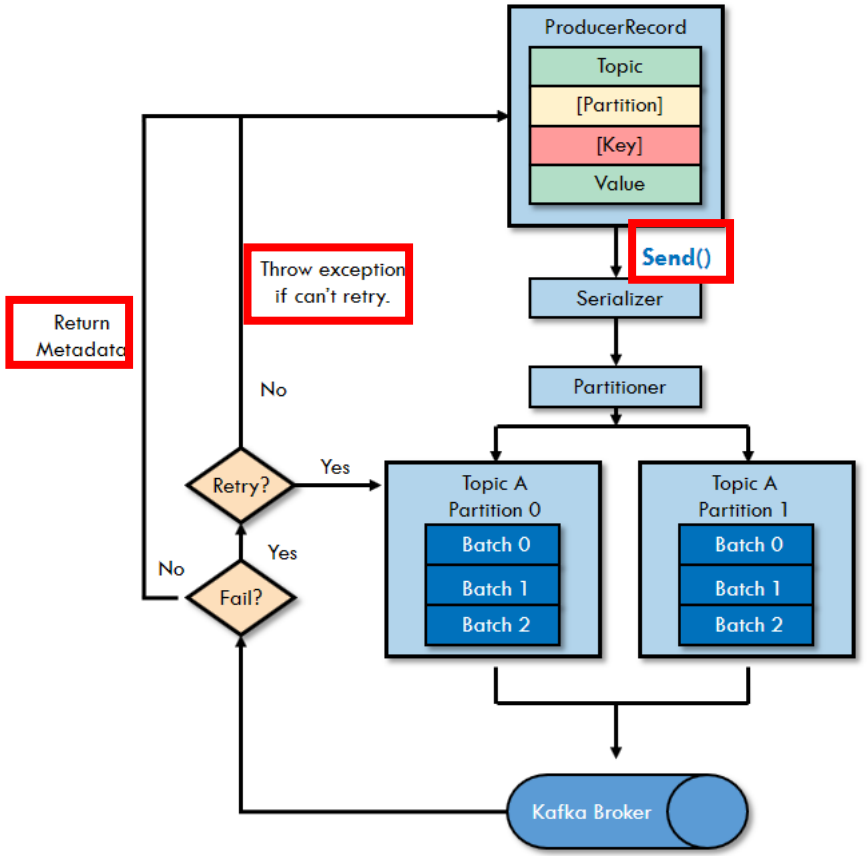
```
// ** 示範: Synchronous Send **
// 在以下的"send().get()"的同步發佈訊息的過程, 透過同步我們可以取得由Broker回覆訊息發佈的ack結果
// 由於這種方法是一筆一筆的送與等待Broker的回應。因此這種方法的throughput最差 (通常不會用Kafka來做這種事)

for(int i=0; i<msgCount; i++) {
    // Broker回覆訊息包含了訊息落在topic的那個partition及offset
    RecordMetadata recordMetadata = producer.send(new ProducerRecord<>(topicName,  key: ""+i,  value: "msg_"+i)).get();

    if((i+1)%10000==0) {
        // 為了不讓打印訊息拖慢速度, 我們每1萬打印一筆recordMetata來看
        System.out.println((i+1) + " messages sent!");
        System.out.println("Topic:Partition:Offset: [" + recordMetadata.topic() + "]:["
                + recordMetadata.partition() + "]:["
                + recordMetadata.offset() + "]");
    }
}
```



Three primary methods of sending messages
Asynchronous send

- Producer_03_Async
  - BrokerAckCallback

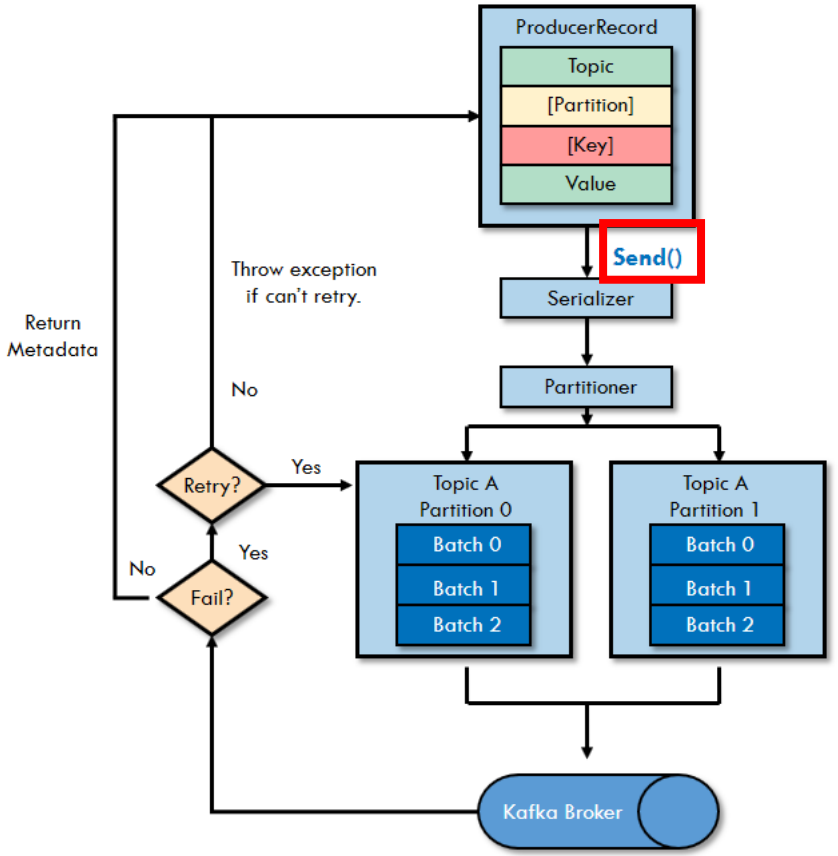
```
// ** 示範: Asynchronous Send **
// 透過一個我們自己定義的Callback函式我們可以非同步地取得由Broker回覆訊息發佈的ack結果
// 這種方法可以取得Broker回覆訊息發佈的ack結果, 同時又可以取得好的throughput (建議的作法)

AtomicInteger atomicInteger = new AtomicInteger();
CountDownLatch countDownLatch = new CountDownLatch(msgCount);

for(int i=0; i<msgCount; i++) {
    // 回呼函式會在Broker送回ack的時候自動去呼叫
    producer.send(new ProducerRecord<>(topicName,  key: ""+i,  value: "msg_" + i),
            new BrokerAckCallback(atomicInteger, countDownLatch));
}
```



Three primary methods of sending messages
Asynchronous send

- Producer_03_Async
  - BrokerAckCallback

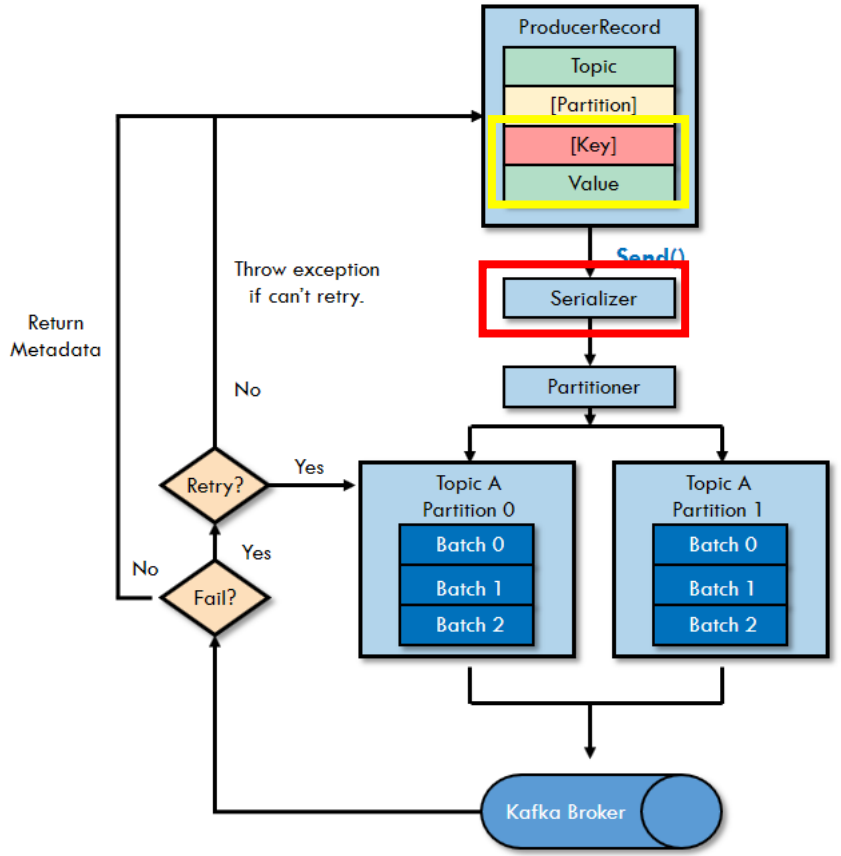
```
public class BrokerAckCallback implements Callback {
    // 使用AtomicInteger來做為非同步的counter, 可以避免blocking及contention
    AtomicInteger atomicInteger;
    // 使用CountdownLatch來讓主線程知道所有的訊息都已經處理完了
    CountDownLatch countDownLatch;

    public BrokerAckCallback(AtomicInteger atomicInteger, CountDownLatch countDownLatch) {
        this.atomicInteger = atomicInteger;
        this.countDownLatch = countDownLatch;
    }

    @Override
    public void onCompletion(RecordMetadata recordMetadata, Exception exception) {
        // 增加1再取出
        int counter = atomicInteger.incrementAndGet();
        if(counter%100000==0) {
            // 為了不讓打印訊息拖慢速度, 我們每10萬打印一筆recordMetata來看
            System.out.println(counter + " messages sent!");
            System.out.println("Topic:Partition:Offset: [" + recordMetadata.topic() + "]:["
                    + recordMetadata.partition() + "]:["
                    + recordMetadata.offset() + "]");
        }
        this.countDownLatch.countDown(); // 倒數
    }
}
```



Configuring Producers - Serializers

Kafka lets us publish and subscribe to streams of records and the records can be of any type (JSON, String, POJO, etc.)

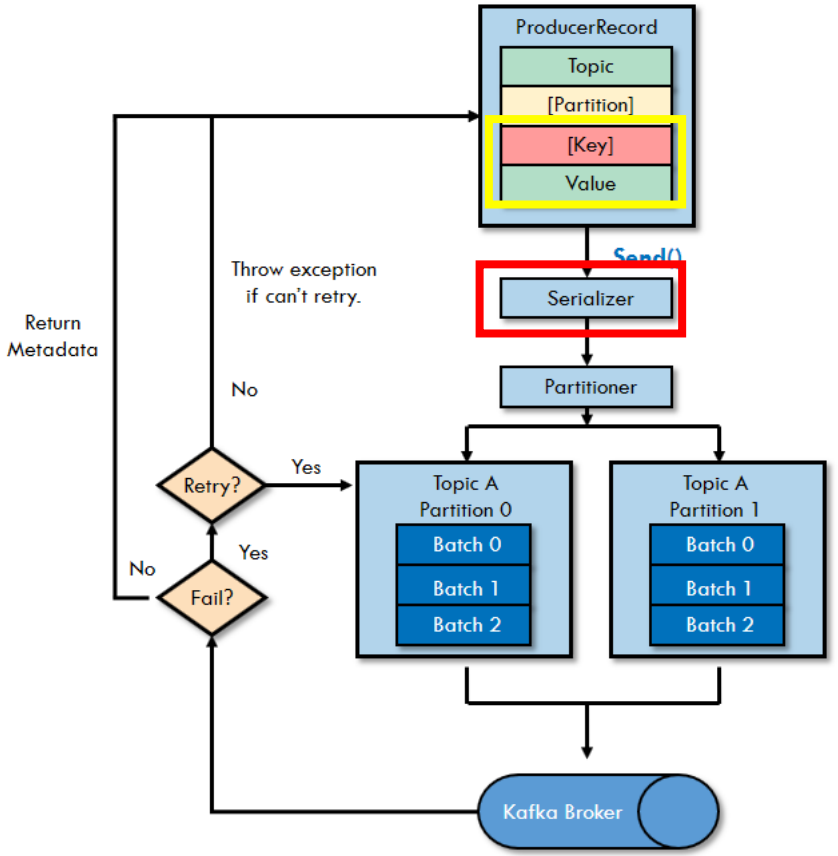
- **key.serializer & value.serializer**
  - **String – StringSerializer**
    - **JSON, XML, Delimited Text, etc..**
  - Integer - IntegerSerializer
  - Long - LongSerializer
  - Double - DoubleSerializer
  - ByteBuffer - ByteBufferSerializer
  - **byte[] – ByteArraySerializer**
    - **Avro, ProtoBuf, etc..**



Configuring Producers - Serializers

- Producer_04_Json
  - **key.serializer & value.serializer**
    - **String – StringSerializer (JSON)**

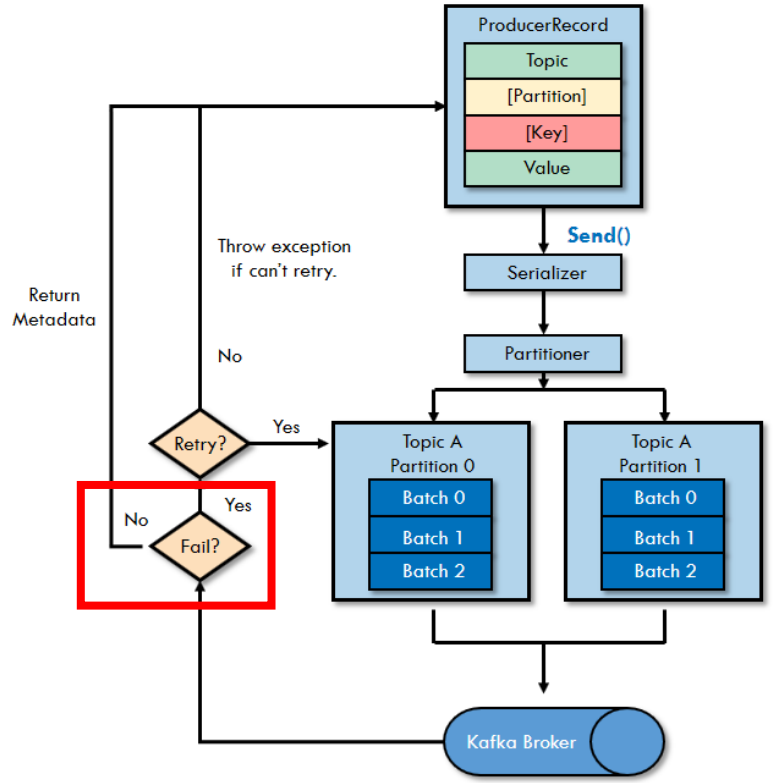
```
props.put("bootstrap.servers", "localhost:9092"); // Kafka集群在那裡?
props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer"); // 指定msgKey的序列化器
props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer"); // 指定msgValue的序列化器

// 一個用來將DTO物件轉換成(JSON String)的物件 <-- 重要
ObjectMapper om = new ObjectMapper();

for(int i=0; i<msgCount; i++) {
    // 讓我們產生假的Employee資料
    Employee employee = new Employee( id: "empid_"+i, firstName: "fn_"+i, lastName: "ln_"+i,
            deptId: "deptid_"+i%10, new Date(), wage: 100000*random.nextFloat(), random.nextBoolean()

    // 把Employee轉換成JSON字串(String)
    String employeeJSON = om.writeValueAsString(employee);

    // 送出訊息
    producer.send(new ProducerRecord<>(topicName, employee.getId(), employeeJSON),
            new BrokerAckCallback(atomicInteger, countDownLatch)); // 回呼函式會在Broker送回ack的時候自動去呼叫
}
```

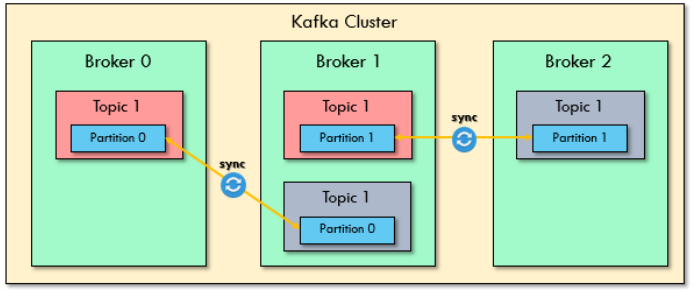


Configuring Producers

- **acks=0**, the producer will not wait for a reply from the broker before assuming the message was sent successfully.
- **acks=1**, the producer will receive a success response from the broker the moment the leader replica received the message.
- **acks=all**, the producer will receive a success response from the broker once all in-sync replicas received the message.

```
props.put("acks","all");
```

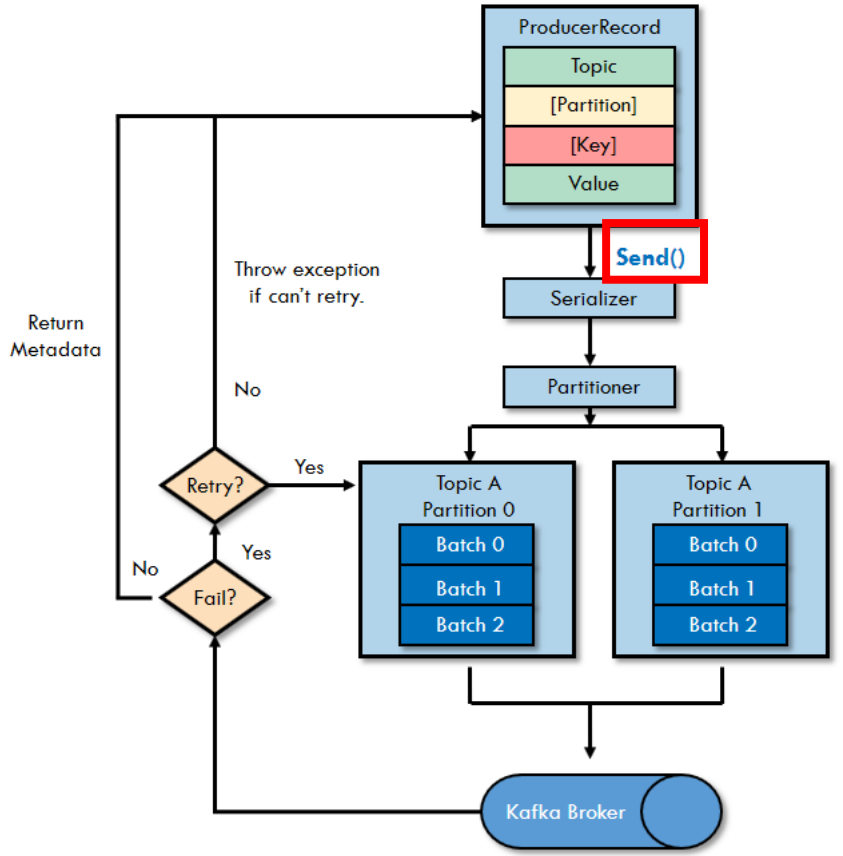


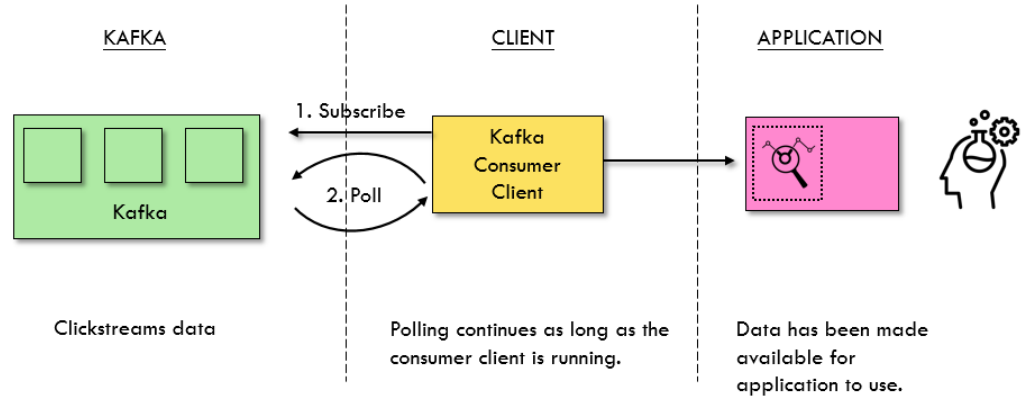


Configuring Producers

- **enable.idempotence**
  - When set to true, the producer will ensure that messages are successfully produced exactly once and in the original produce order.

```
props.put("enable.idempotence", "true");
```

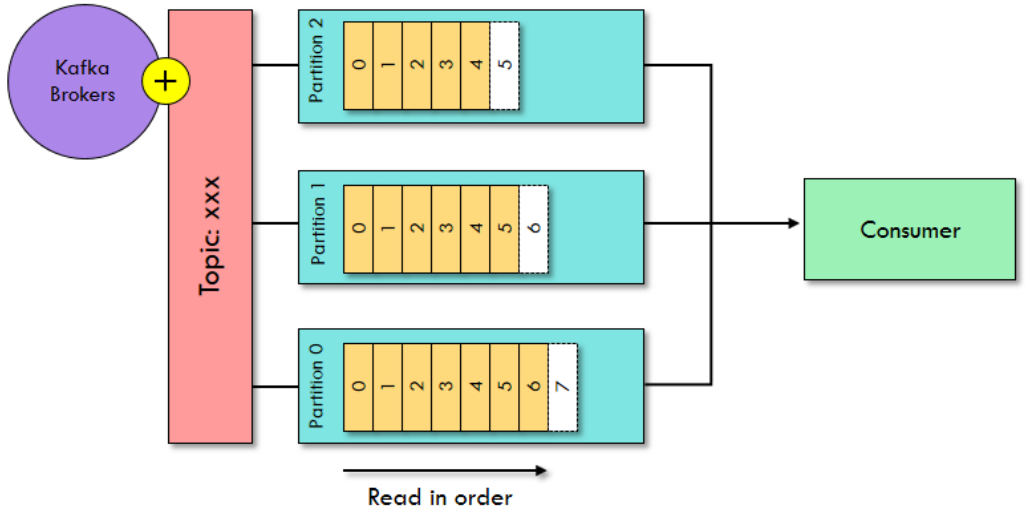




Kafka Consumers: Reading Messages from Kafka

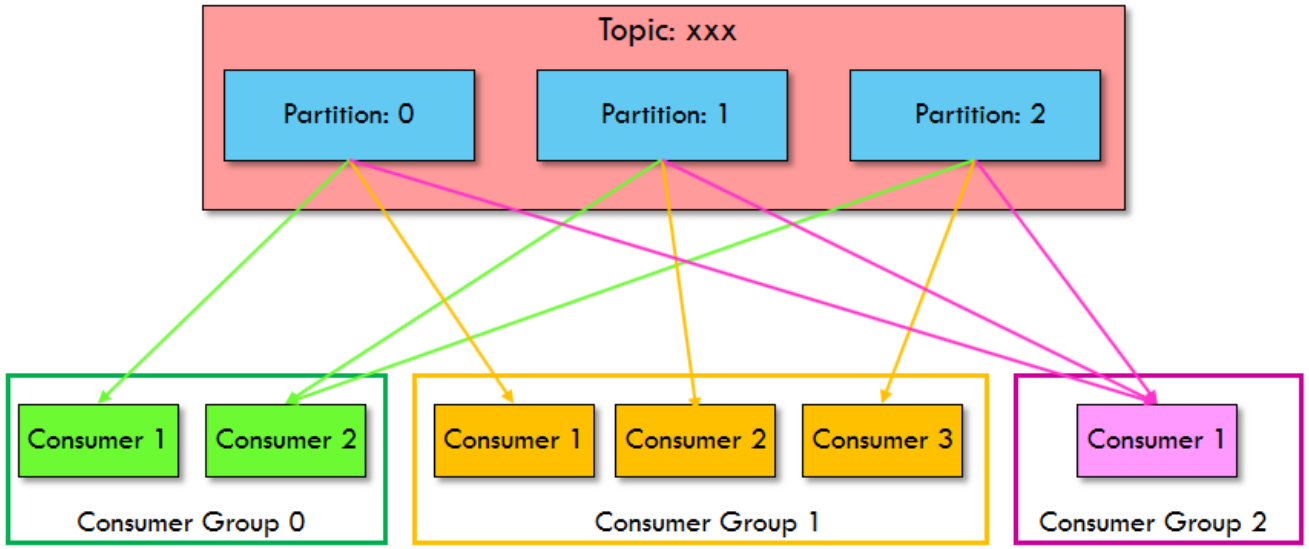
Consumers

- **Consumers** read data from a topic
- They only have to specify the topic name and one broker to connect to, and Kafka will automatically take care of pulling the data from the right brokers
- Data is read in order **for each partitions**



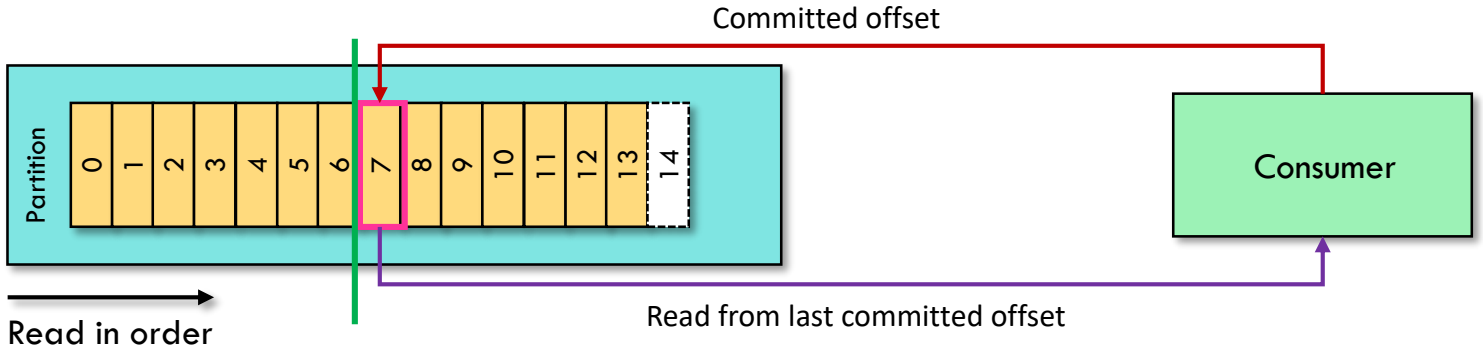
Consumer Groups

- Consumers read data in **consumer groups**
- Each consumer within a group reads from exclusive partitions
- You cannot have more consumers than partitions (otherwise some will be inactive)



Consumer Offsets

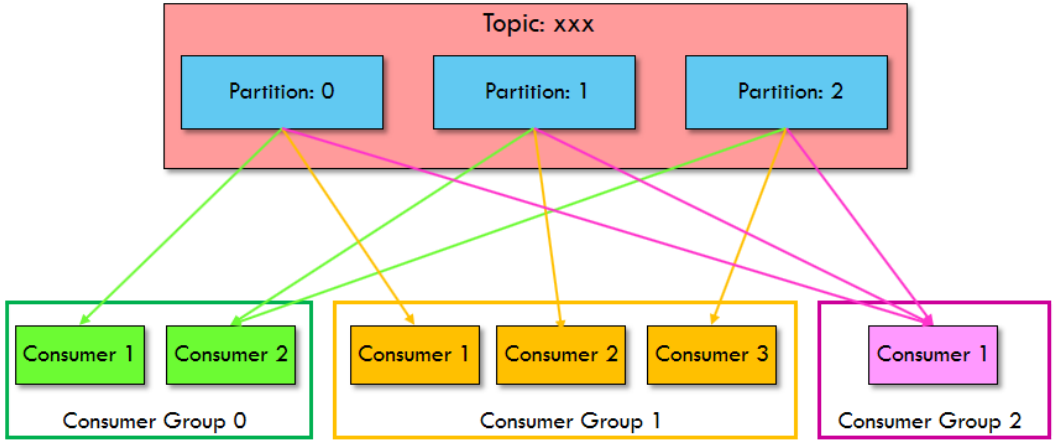
- Kafka stores the offsets at which a **consumer group** has been reading
- The **offsets** commit live in a Kafka topic named "**__consumer_offsets**"
- When a consumer has processed data received some Kafka, it should be **committing** the **offsets**
- If a consumer process dies, it will be able to read back from where it left off thanks to consumer offsets!



Configuring Consumers

- **group.id**
  - Kafka consumers are typically part of a **consumer group**. When multiple consumers are subscribed to a topic and belong to the same consumer group, each consumer in the group will receive messages from a different subset of the partitions in the topic.

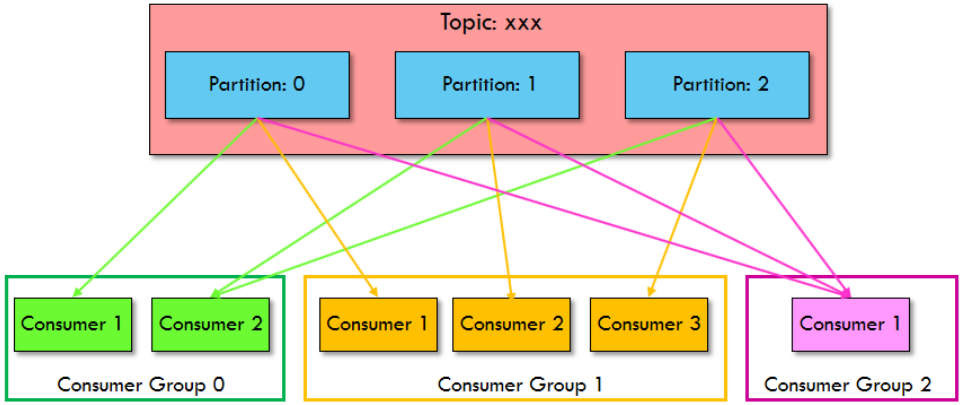
```
Properties props = new Properties();
props.put("bootstrap.servers", "broker1:9092,broker2:9092");
props.put("group.id", "CountryCounter");
props.put("key.deserializer",
  "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer",
  "org.apache.kafka.common.serialization.StringDeserializer");

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



Consumers and Consumer Groups

Create a topic with 4 partitions

```
$ kafka-topics
    --create
    --zookeeper localhost:2181
    --replication-factor 1 --partitions 4
    --topic ak03.fourpartition
```



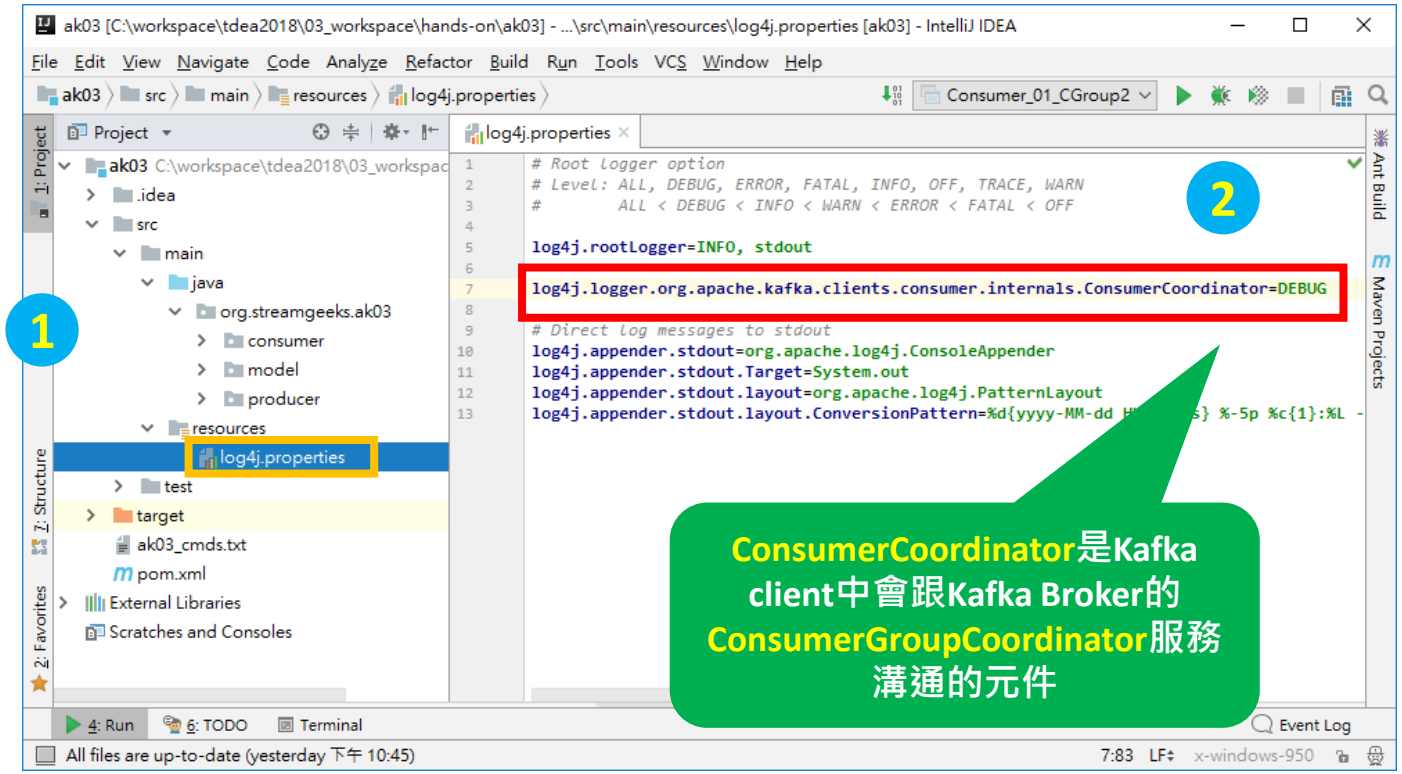

```
root@kafka:/# kafka-topics --create --zookeeper zookeeper:2181 \
>    --replication-factor 1 \
>    --partitions 4 \
>    --topic ak03.fourpartition
WARNING: Due to limitations in metric names, topics with a period ('.') or under
score ('_') could collide. To avoid issues it is best to use either, but not bot
h.
Created topic "ak03.fourpartition".
```

在container裡頭的zookeeper服務的Hostname

Topic is created!

Consumers and Consumer Groups

Change ConsumerCoordinator Log Level



Consumers and Consumer Groups

Producer_05_CGroup : Publish Event:

```java
/**
 * 示範: 用來持續產生event來給Consumer_01_CGroup進行Rebalance的演示
 */
public class Producer_05_CGroup {
    public static void main(String[] args) {
        // 步驟1. 設定要連線到Kafka集群的相關設定
        Properties props = new Properties();
        props.put("bootstrap.servers", "localhost:9092"); // Kafka集群在那裡?
        props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer"); // 指定msgKey的序列化器
        props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer"); // 指定msgValue的序列化器
        // 步驟2. 產生一個Kafka的Producer的實例
        Producer<String, String> producer = new KafkaProducer<>(props);
        // 步驟3. 指定想要發佈訊息的topic名稱
        String topicName = "ak03.fourpartition";
        int msgCount = 10000; // 1萬筆
        try {
            System.out.println("Start sending messages ...");
            // 步驟4. 產生要發佈到Kafka的訊息 (把訊息封裝進一個ProducerRecord的實例中)
            for(int i=0; i<msgCount; i++) {
                producer.send(new ProducerRecord<>(topicName, key: ""+i, value: "msg_"+i));
                Thread.sleep( millis: 3000); // 註執行緒停個3秒
            }
        } catch (Exception e) {
            // 錯誤處理
            e.printStackTrace();
        }
        // 步驟5. 關掉Producer實例的連線
        producer.close();
        System.out.println("Message sending completed!");
    }
}
```

這個Producer的目的是持續發佈有序列號的訊息來觀察ConsuermGroup的概念!

每3秒發佈一筆訊息 !

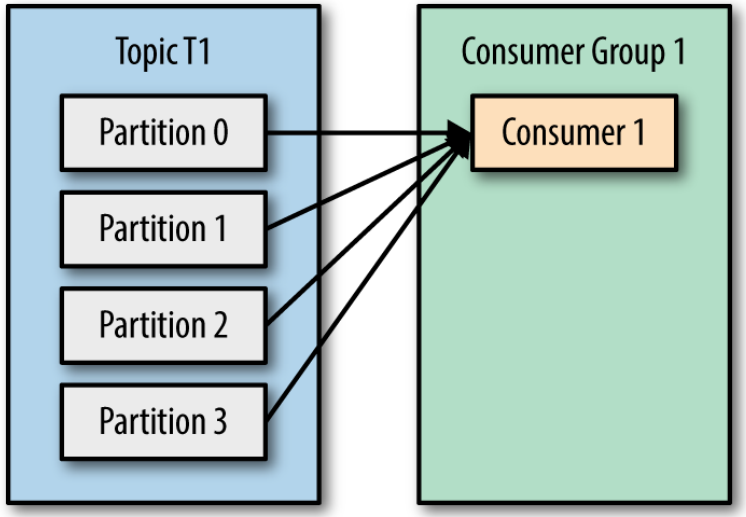
Consumers and Consumer Groups

- Producer_05_CGroup

```
for(int i=0; i<msgCount; i++) {
    producer.send(new ProducerRecord<>(topicName,  key: ""+i,  value: "msg_"+i));

    Thread.sleep( millis: 3000); // 讓主執行緒停個3秒
}
```

- Consumer_01_Cgroup
  - One Consumer instance

One Consumer with four partitions



跑第1個instance, 並觀察DEBUG的log

```
INFO  ConsumerCoordinator:411                    [Consumer clientId=consumer-1, groupId=CG1]

Revoking previously assigned partitions []
Setting newly assigned partitions [ak03.fourpartition-0, ak03.fourpartition-2, ak03.fourpartition-1, ak03.fourpartition-3]
```

Consumers and Consumer Groups

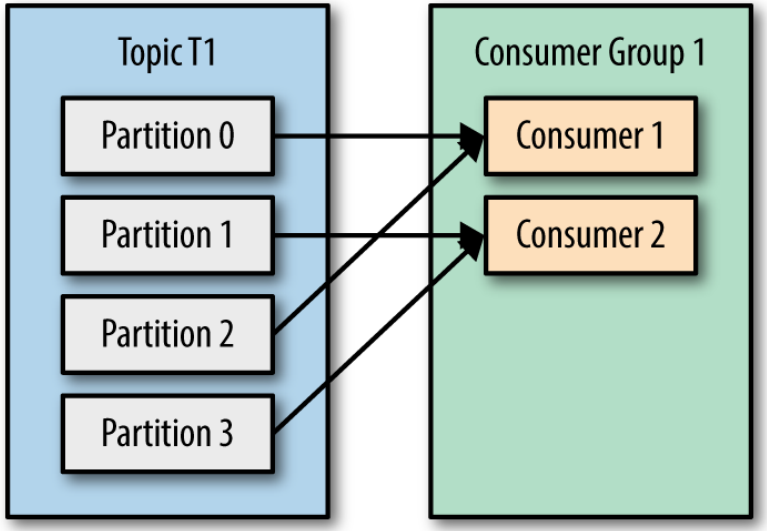
- Producer_05_CGroup

```
for(int i=0; i<msgCount; i++) {
    producer.send(new ProducerRecord<>(topicName,  key: ""+i,  value: "msg_"+i));

    Thread.sleep( millis: 3000); // 讓主執行緒停個3秒
}
```

- Consumer_01_Cgroup
  - Two Consumer instances

跑第2個instance, 並觀察DEBUG的log

Two Consumer with four partitions




```
[Consumer clientId=consumer-1, groupId=CG1] Revoking previously assigned partitions []
[Consumer clientId=consumer-1, groupId=CG1] Setting newly assigned partitions [ak03.fourpartition-0, ak03.fourpartition-2, ak03.fourpartition-1, ak03.fourpartition-3]

[Consumer clientId=consumer-1, groupId=CG1] Revoking previously assigned partitions [ak03.fourpartition-0, ak03.fourpartition-2, ak03.fourpartition-1, ak03.fourpartition-3]
[Consumer clientId=consumer-1, groupId=CG1] Setting newly assigned partitions [ak03.fourpartition-0, ak03.fourpartition-1]



[Consumer clientId=consumer-1, groupId=CG1] Revoking previously assigned partitions []
[Consumer clientId=consumer-1, groupId=CG1] Setting newly assigned partitions [ak03.fourpartition-2, ak03.fourpartition-3]
```

Consumers and Consumer Groups

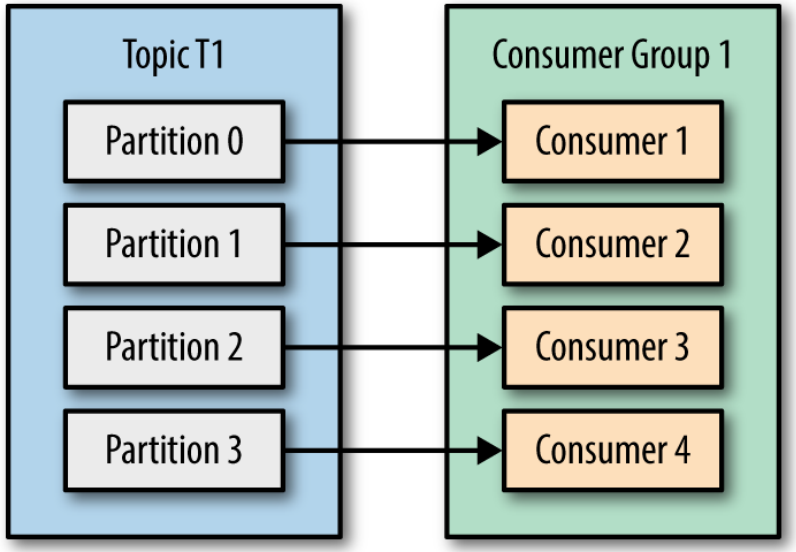
- Producer_05_CGroup

```
for(int i=0; i<msgCount; i++) {
    producer.send(new ProducerRecord<>(topicName,  key: ""+i,  value: "msg_"+i));

    Thread.sleep( millis: 3000); // 讓主執行緒停個3秒
}
```

- Consumer_01_Cgroup
  - Four Consumer instances

跑第4個instance, 並觀察DEBUG的log

Three Consumer with four partitions




```
[Consumer clientId=consumer-1, groupId=CG1] Setting newly assigned partitions [ak03.fourpartition-3]
[Consumer clientId=consumer-1, groupId=CG1] Setting newly assigned partitions [ak03.fourpartition-1]
[Consumer clientId=consumer-1, groupId=CG1] Setting newly assigned partitions [ak03.fourpartition-0]
[Consumer clientId=consumer-1, groupId=CG1] Setting newly assigned partitions [ak03.fourpartition-2]
```

Consumers and Consumer Groups

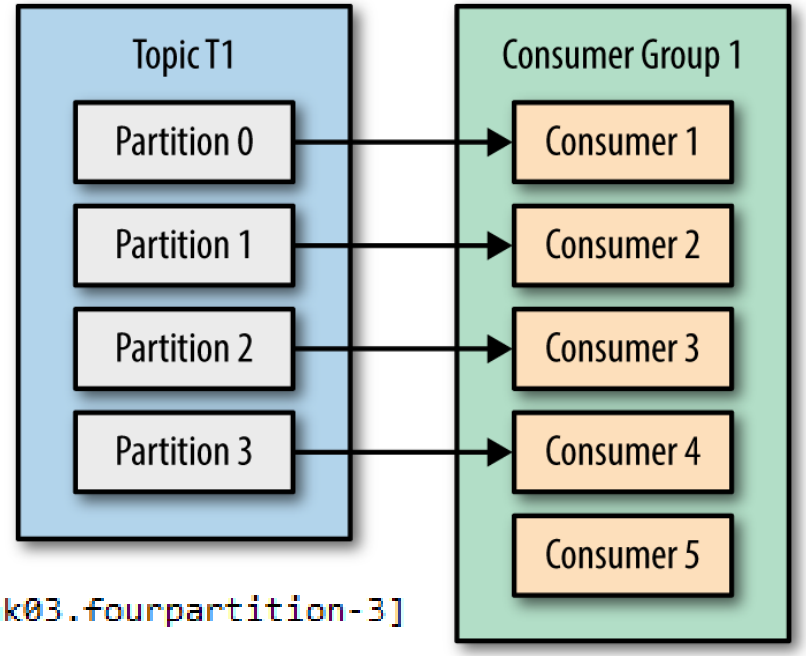
- Producer_05_CGroup

```
for(int i=0; i<msgCount; i++) {
    producer.send(new ProducerRecord<>(topicName,  key: ""+i,  value: "msg_"+i));

    Thread.sleep( millis: 3000); // 讓主執行緒停個3秒
}
```

- Consumer_01_Cgroup
  - Five Consumer instances

跑第5個instance, 並觀察DEBUG的log

Five Consumer with four partitions




```
[Consumer clientId=consumer-1, groupId=CG1] Setting newly assigned partitions [ak03.fourpartition-3]
[Consumer clientId=consumer-1, groupId=CG1] Setting newly assigned partitions [ak03.fourpartition-1]
[Consumer clientId=consumer-1, groupId=CG1] Setting newly assigned partitions [ak03.fourpartition-0]
[Consumer clientId=consumer-1, groupId=CG1] Setting newly assigned partitions []
[Consumer clientId=consumer-1, groupId=CG1] Setting newly assigned partitions [ak03.fourpartition-2]
```

What is Rebalance?

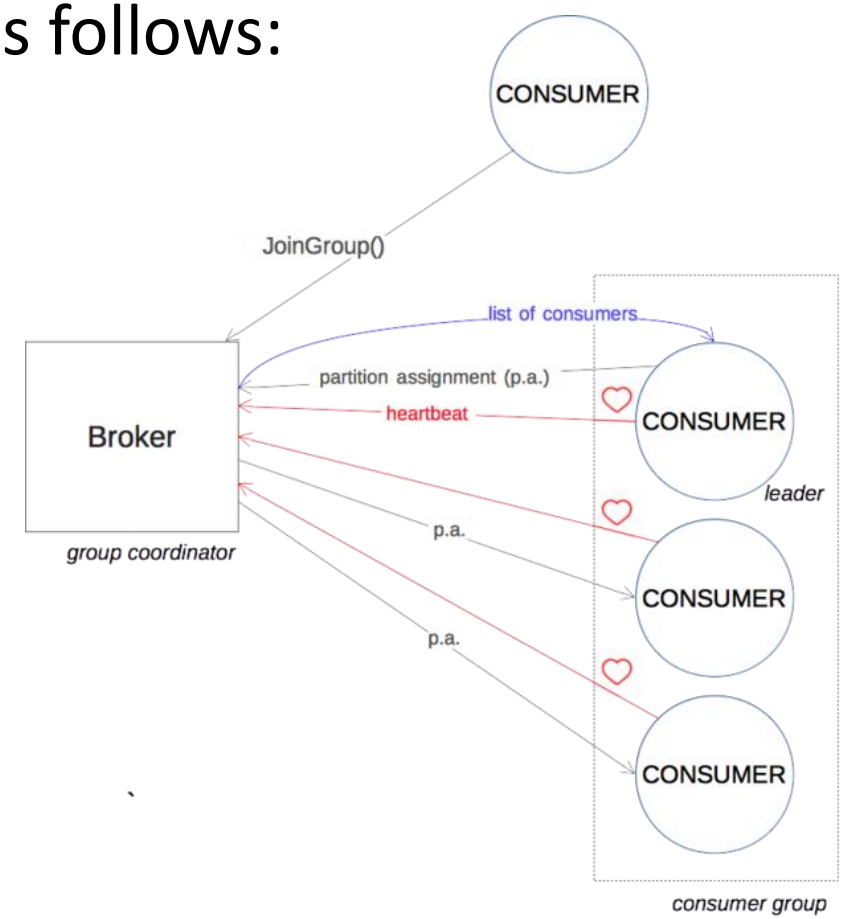
- Every consumer in a **consumer group** is assigned one or more topic partitions exclusively, and **Rebalance** is the re-assignment of partition ownership among consumers.
- A Rebalance happens when:
  - a consumer JOINS the group
  - a consumer SHUTS DOWN cleanly
  - a consumer is considered DEAD by the group coordinator. This may happen after a crash or when the consumer is busy with a long-running processing
  - new partitions are added

Rebalance process

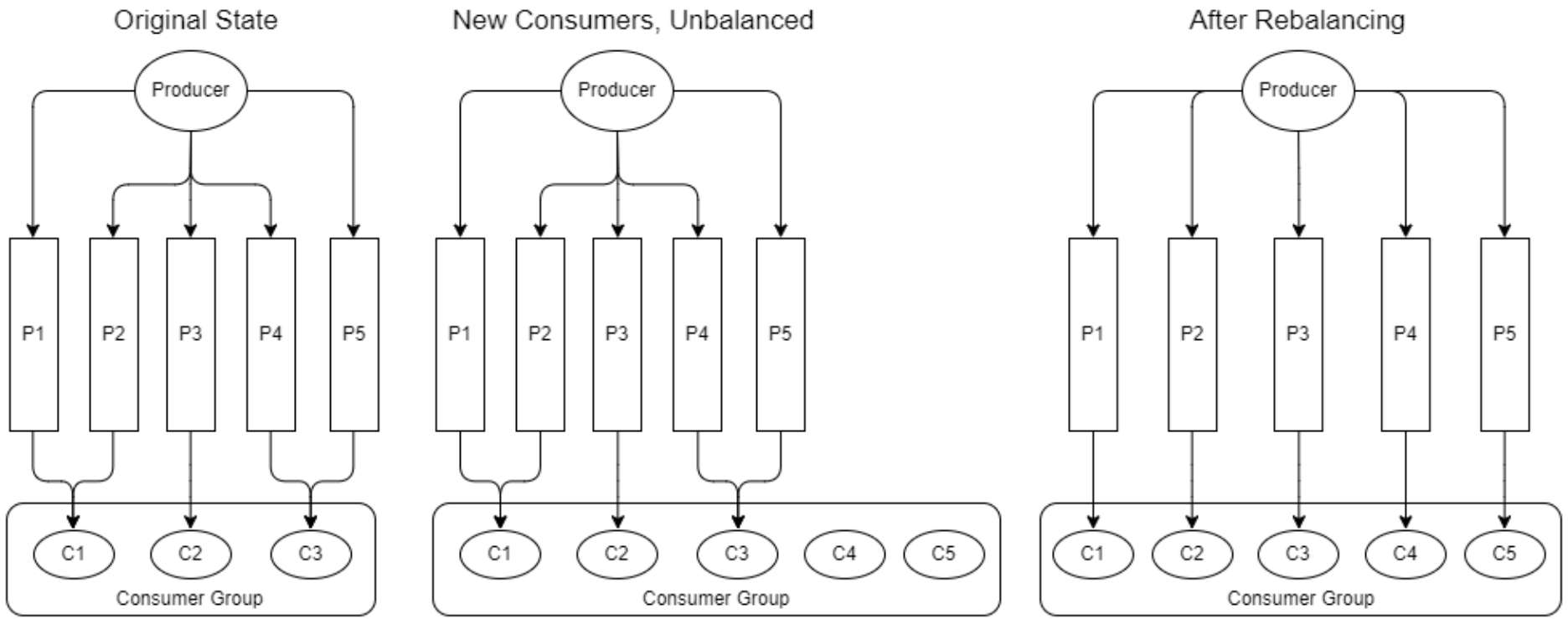
Rebalance can be more or less described as follows:

- The leader receives a list of all consumers in the group from the group coordinator and is responsible for assigning a subset of partitions to each consumer.

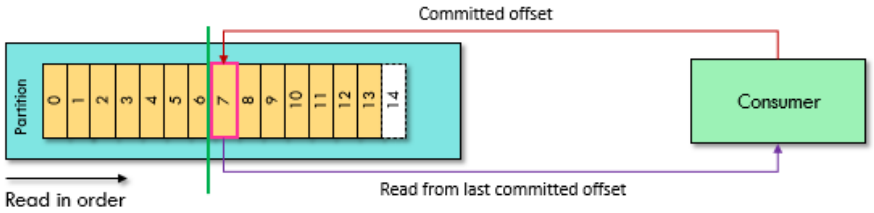
- After deciding on the partition assignment the group leader sends the list of assignments to the group coordinator, which sends this information to all the consumers.



Rebalance process



Commits and Offsets



- Whenever we call **poll()**, broker returns records that consumers in our group have not read yet.
- This means that kafka client tracking which records were read by a consumer of the group.
- We call the action of updating the current position in the partition a **commit**.

```
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records)
    {
        System.out.printf("topic = %s, partition = %s, offset =
          %d, customer = %s, country = %s\n",
             record.topic(), record.partition(),
```

Commits and Offsets

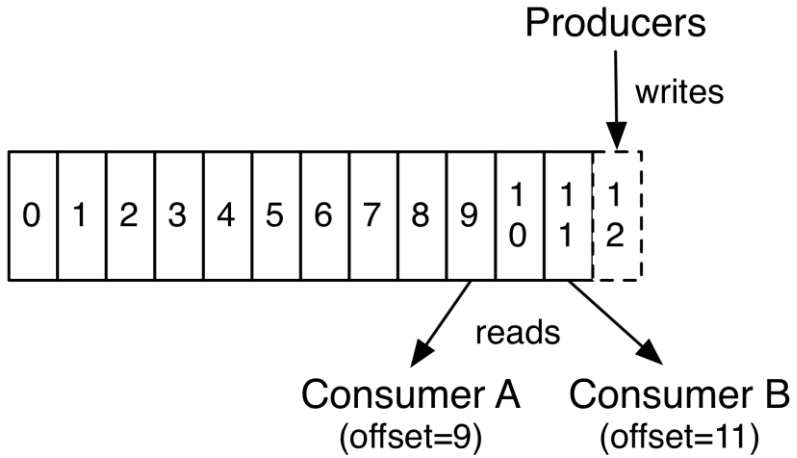
- How does a consumer commit an offset?
  - Kafka client produces a message to Kafka, to a special **__consumer_offsets** topic, with the committed offset for each partition.
  - If a consumer crashes or a new consumer joins the consumer group, it will trigger a **partition rebalance**.
  - After a **rebalance**, each consumer may be assigned a new set of partitions than the one it processed before.
  - The consumer will read the latest committed offset of each partition and continue from there.

Commits and Offsets

- Managing offsets has a big impact on the client application.
- The Kafka Consumer API provides multiple ways of committing offsets:
  - Automatic Commit
  - Synchronous Commit (previous **poll()** offset)
  - Asynchronous Commit (previous **poll()** offset)
  - Combining Synchronous and Asynchronous Commits (previous **poll()** offset)
  - Commit Specified Offset

Configuring Consumers

- **auto.offset.reset**
  - This property controls the behavior of the consumer when it starts reading a partition for which it doesn't have a committed offset.
  - The default is "**latest**," which means that lacking a valid offset, the consumer will start reading from the newest records
  - The alternative is "**earliest**," which means that lacking a valid offset, the consumer will read all the data in the partition, starting from the very beginning.



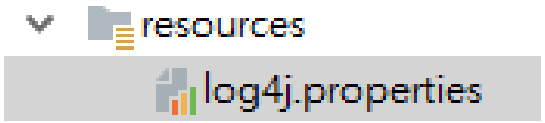
Commits and Offsets: Automatic Commit

- Configure below configurations:
  - **enable.auto.commit=true**
  - **auto.commit.interval.ms=5000 (default)**
- Every **five** seconds the consumer will commit the largest offset your client received from poll().
- Whenever you poll, the consumer checks if it is time to commit, and if it is, it will commit the offsets it returned in the last poll.

Commits and Offsets: **Automatic Commit**

- Consumer_02_AutoCommit

```
props.put("auto.offset.reset", "earliest"); // 是否從這個ConsumerGroup尚未讀取的partition/offset開始讀

props.put("enable.auto.commit", "true");
props.put("auto.commit.interval.ms", "5000");
```




```
log4j.logger.org.apache.kafka.clients.consumer.internals.ConsumerCoordinator=DEBUG
```

```
Sending asynchronous auto-commit of offsets {ak03.fourpartition-0=OffsetAndMetadata{offset=85, metadata=''}, 
Committed offset 85 for partition ak03.fourpartition-0
Committed offset 65 for partition ak03.fourpartition-2
Committed offset 63 for partition ak03.fourpartition-1
Committed offset 73 for partition ak03.fourpartition-3
Completed asynchronous auto-commit of offsets {ak03.fourpartition-0=OffsetAndMetadata{offset=85, metadata=''}
```

Commits and Offsets: Automatic Commit

- With **auto-commit** enabled, a call to poll() *will always commit the last offset returned by the previous poll*. It doesn't know which events were actually processed, so it is critical to always process all the events returned by poll() before calling poll() again.
- Automatic commits are convenient, but they don't give developers enough control to avoid duplicate messages.

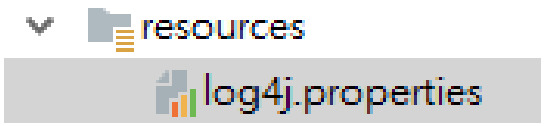
Commits and Offsets: Manual Commit

- The consumer API has the option of committing the current offset at a point that makes sense to the application developer rather than based on a timer.
- Configure below configurations:
  - **auto.commit.offset=false**
- Offsets will only be committed when the application explicitly chooses to do so.
- The simplest and most reliable of the commit APIs is **commitSync()**.

Commits and Offsets: Sync Commit

- Consumer_03_CommitSync

```
props.put("auto.offset.reset", "earliest"); // 是否從這個ConsumerGroup尚未讀取的partition/offset開始讀
props.put("enable.auto.commit", "false");
```

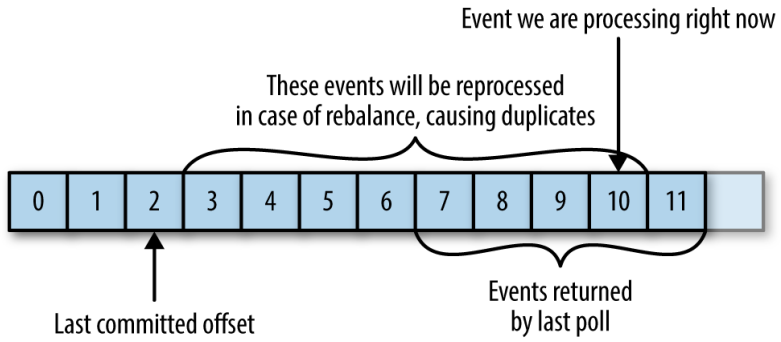



```
log4j.logger.org.apache.kafka.clients.consumer.internals.ConsumerCoordinator=DEBUG
```

```
while (true) {
    // 請求Kafka把新的訊息吐出來
    ConsumerRecords<String, String> records = consumer.poll( timeout: 1000);

    // 如果有任何新的訊息就會進到下面的迭代
    for (ConsumerRecord<String, String> record : records){
        // ** 在這裡進行商業邏輯與訊息處理 **
        // ....
    }
    try {
        consumer.commitSync(); // <-- Commit last Poll()的offset
    } catch (CommitFailedException e) {
        // 錯誤處理
        System.out.println("commit failed");
        e.printStackTrace();
    }
```

Commits and Offsets: Sync Commit

- This **commitSync()** will commit the latest offset returned by **poll()** and return once the offset is committed, throwing an exception if commit fails for some reason.
- Make sure you call **commitSync()** after you are done processing all the records in the collection.
- When **rebalance** is triggered, all the messages from the beginning of the most recent batch until the time of the rebalance will be processed twice.



Commits and Offsets: Asynchronous Commit

- One drawback of manual commit is that the application is ***blocked*** until the broker responds to the commit request.
- Another option is the **asynchronous** commit API. Instead of waiting for the broker to respond to a commit, we just send the request and continue on:

```
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records)
    {
        System.out.printf("topic = %s, partition = %s,
        offset = %d, customer = %s, country = %s\n",
        record.topic(), record.partition(), record.offset(),
        record.key(), record.value());
    }
    consumer.commitAsync(); ❶
}
```

Commits and Offsets: Asynchronous Commit

- Consumer_04_CommitAsync

```
props.put("auto.offset.reset", "earliest"); // 是否從這個ConsumerGroup尚未讀取的partition/offset開始讀
props.put("enable.auto.commit", "false");
```

resources

log4j.properties

```
log4j.logger.org.apache.kafka.clients.consumer.internals.ConsumerCoordinator=DEBUG
```

```
while (true) {
    // 請求Kafka把新的訊息吐出來
    ConsumerRecords<String, String> records = consumer.poll( timeout: 1000);
    // 如果有任何新的訊息就會進到下面的迭代
    for (ConsumerRecord<String, String> record : records){
        // ** 在這裡進行商業邏輯與訊息處理 **
        // ..
    }
    consumer.commitAsync(new OffsetCommitCallback() {
        @Override
        public void onComplete(Map<TopicPartition, OffsetAndMetadata> offsets, Exception e) {
            if (e != null) {
                // 錯誤處理
                System.out.println("commit failed");
                e.printStackTrace();
            }
        }
    });
```

Commits and Offsets: Asynchronous Commit

- The drawback is that while **commitSync()** will retry the commit until it either succeeds or encounters a non-retriable failure
- **commitAsync()** will not retry.
- **commitAsync()** gives you an option to pass in a callback that will be triggered when the broker responds.

```
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records) {
        System.out.printf("topic = %s, partition = %s,
        offset = %d, customer = %s, country = %s\n",
        record.topic(), record.partition(), record.offset(),
        record.key(), record.value());
    }
    consumer.commitAsync(new OffsetCommitCallback() {
        public void onComplete(Map<TopicPartition,
        OffsetAndMetadata> offsets, Exception exception) {
            if (e != null)
                log.error("Commit failed for offsets {}", offsets, e);
        }
      }); ❶
```

Commits and Offsets: Combining Sync and Async Commits

```
try {
    while (true) {
        ConsumerRecords<String, String> records = consumer.poll(100);
        for (ConsumerRecord<String, String> record : records) {
            System.out.printf("topic = %s, partition = %s, offset = %d,
            customer = %s, country = %s\n",
            record.topic(), record.partition(),
            record.offset(), record.key(), record.value());
        }
        consumer.commitAsync(); ❶
    }
} catch (Exception e) {
    log.error("Unexpected error", e);
} finally {
    try {
        consumer.commitSync(); ❷
    } finally {
        consumer.close();
    }
}
```

Commits and Offsets: Combining Sync and Async Commits

- Consumer_05_CommitSyncAsync

```
// 步驟5. 持續的拉取Kafka有進來的訊息
try {
    System.out.println("Start listen incoming messages ...");

    while (true) {
        // 請求Kafka把新的訊息吐出來
        ConsumerRecords<String, String> records = consumer.poll( timeout: 1000);

        // 如果有任何新的訊息就會進到下面的迭代
        for (ConsumerRecord<String, String> record : records) {
            // ** 在這裡進行商業邏輯與訊息處理 **
            // ...
        }
        consumer.commitAsync();
    }
} catch(Exception e) {
    // log.error("Unexpected error", e);
    e.printStackTrace();
} finally {
    // 步驟6. 如果收到結束程式的訊號時關掉Consumer實例的連線
    try {
        consumer.commitSync();
    } finally {
        consumer.close();
    }
    System.out.println("Stop listen incoming messages");
}
```

Commits and Offsets: Commit Specified Offset

- Committing the latest offset only allows you to commit as often as you finish processing batches.
- What if **poll()** returns a huge batch and you want to commit offsets in the middle of the batch to avoid having to process all those rows again if a rebalance occurs?
- You can't just call **commitSync()** or **commitAsync()** —this will commit the last offset returned, which you didn't get to process yet.
- Consumer API allows you to call commitSync() and commitAsync() and pass a map of **partitions** and **offsets** that you wish to commit.

Commits and Offsets: **Commit Specified Offset**

```
private Map<TopicPartition, OffsetAndMetadata> currentOffsets =
    new HashMap<>(); ❶
int count = 0;

....

while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records)
    {
        System.out.printf("topic = %s, partition = %s, offset = %d,
        customer = %s, country = %s\n",
        record.topic(), record.partition(), record.offset(),
        record.key(), record.value()); ❷
        currentOffsets.put(new TopicPartition(record.topic(),
        record.partition()), new
        OffsetAndMetadata(record.offset()+1, "no metadata")); ❸
        if (count % 1000 == 0)    ❹
            consumer.commitAsync(currentOffsets, null); ❺
        count++;
    }
}
```

Commits and Offsets: Commit Specified Offset

- Consumer_06_CommitSpecified

```java
// 步驟5. 產生一個紀錄TopicPartition與Offset的容器
Map<TopicPartition, OffsetAndMetadata> currentOffsets = new HashMap<>();
// 步驟6. 持續的拉取Kafka有進來的訊息
int recordCount = 0;
try {
    System.out.println("Start listen incoming messages ...");
    while (true) {
        // 請求Kafka把新的訊息吐出來
        ConsumerRecords<String, String> records = consumer.poll( timeout: 1000);
        // 如果有任何新的訊息就會進到下面的迭代
        for (ConsumerRecord<String, String> record : records) {
            // ** 在這裡進行商業邏輯與訊息處理 **
            // 取出相關的metadata
            String topic = record.topic();
            int partition = record.partition();
            long offset = record.offset();
            TimestampType timestampType = record.timestampType();
            long timestamp = record.timestamp();
            // 取出msgKey與msgValue
            String msgKey = record.key();
            String msgValue = record.value();
            // 秀出metadata與msgKey & msgValue訊息
            System.out.println(topic + "-" + partition + "-" + offset + " : (" + record.key() + ", " + record.value() + ")");
            // 紀錄現在的TopicPartition與Offset
            currentOffsets.put(new TopicPartition(topic, partition), new OffsetAndMetadata( offset: offset+1,  metadata: "no metadata"));
            // 決定何時要commit
            if (recordCount % 1000 == 0) {
                Map<TopicPartition, OffsetAndMetadata> offsetToCommits = copyHashmap(currentOffsets);
                consumer.commitAsync(offsetToCommits,  callback: null); // 對特定的offset進行commit
                currentOffsets.clear(); // 清除原有TopicPartition與Offset的紀錄
            }
            recordCount++;
        }
        consumer.commitAsync();
    }
```

Rebalance Listeners

- If you know your consumer is about to lose ownership of a partition, you will want to commit offsets of the last event you've processed.
- The consumer API allows you to run your own code when partitions are added or removed from the consumer.
- You do this by passing a **ConsumerRebalanceListener** when calling the subscribe() method
- **ConsumerRebalanceListener** has two methods you can implement:
  - onPartitionsRevoked
  - onPartitionsAssigned

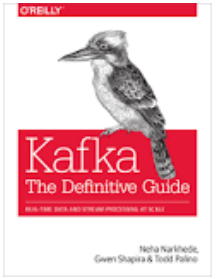
Commits and Offsets: RebalanceListener

- Consumer_07_CommitSpecified

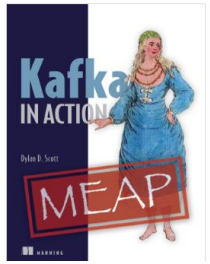
```java
public class RebalanceHandler implements ConsumerRebalanceListener {
    private Map<TopicPartition, OffsetAndMetadata> currentOffsets = new HashMap<>();
    private Consumer consumer;

    public RebalanceHandler(Consumer consumer, Map<TopicPartition, OffsetAndMetadata> currentOffsets){
        this.consumer = consumer;
        this.currentOffsets = currentOffsets;
    }

    // 當Rebalance被觸發後, Consumer被取回被assigned的partitions
    @Override
    public void onPartitionsRevoked(Collection<TopicPartition> partitions) {
        System.out.println("Lost partitions in rebalance. Commiting current Offsets: " + currentOffsets);
        consumer.commitSync(currentOffsets);
    }

    // 當Rebalance被觸發後, Consumer被通知有那些partition被assigned
    @Override
    public void onPartitionsAssigned(Collection<TopicPartition> partitions) {

    }
}
```

Reference

1. Kafka: The Definitive Guide – O'REILLY



2. Kafka In Action - MANNING



3. Learn Apache Kafka for Beginners – Udemy (Stephane Maarek)

4. Confluent Document of Kafka – confluent.io

confluent


The
End