- **Distributed System Kfaka**
  - **問題(目的)與背景**
  - **架構解析**
    - **Kafka Producer’s view**
    - **Kafka Consumer’s view**
    - **Kafka Zookeeper Broker’s view**
  - **技術實作與心得**
  - **評論**
  - **結語**
  - **分工**

Distributed System Kfaka

- 組員
  - 108753208 葉冠宏
  - 109753209 李庭慶
  - 110703065 詹松霖
  - 110753113 張皓博
  - 110753147 王盛泰
  - 110753161 鄭仁傑

問題(目的)與背景

> Briefly explain the core idea of the technology

Kafka建立一個數據中心來作為Message Queue Middleware，它的高產出及低延遲可以有效防止突發性的大數據湧入，並且因為將資訊都傳入Kafka處理，可以適當的解放其他系統的設計靈活度。

> 該分散式系統/技術的應用場景 (Example)，用來說明:
>
> > 在什麼場何用來解決什麼問題?
> > 沒有它又如何?

Kafka 作為一個速度快的分散式event streaming platform，他可以使大量的資料持續的在正確的時間流到正確的位置。其中心為Broker的構造也使Producer與Consumer不必知道互

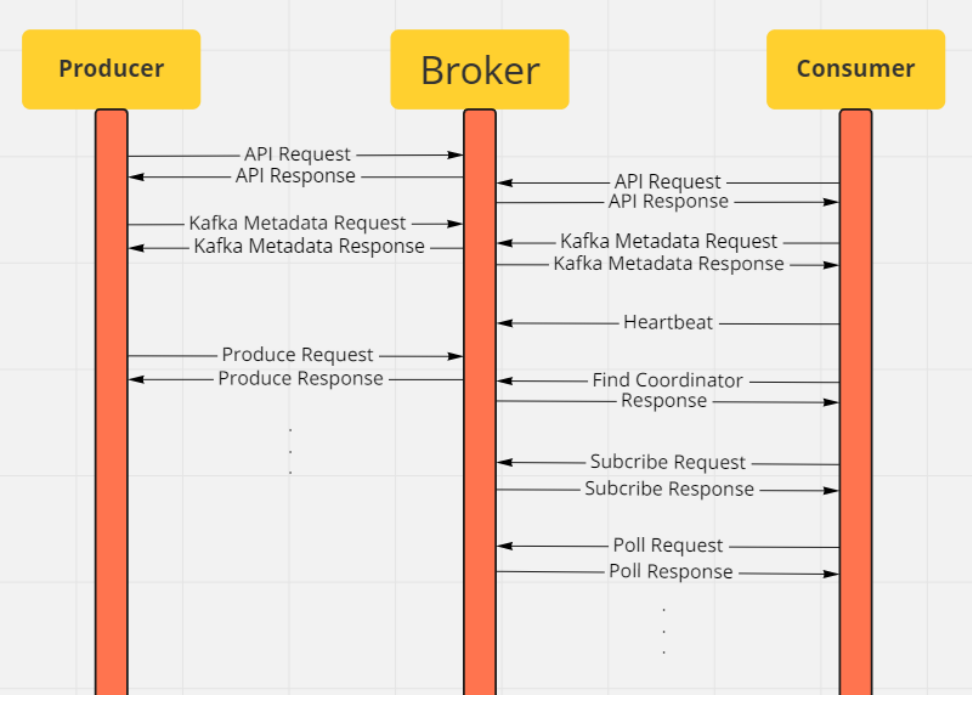
相的位置，可輕鬆替換系統中的組件。很多公司會使用它來做即時的資料串流或反饋。
上述特性使Kafka常用在以下情景

- 即時的付款與金融交易。如證券交易所與銀行
- 追蹤跟監控卡車，貨物。如物流業
- 持續的搜集用戶的操作並做即時的反饋。如netflix的推薦下一部的功能即使用到Kafka。

若沒有中心的Broker構造，分散式系統中的每個組件就必須知道互相的位置，整個系統的耦合度極高。即便自己寫了個簡易的Broker，它掛掉整個系統就癱瘓了。但Kafka能讓各Broker間可以相互協調、取代，使穩定度大幅提升。

架構解析

Kafka中的節點可以分為三種，分別是 Broker, Producer, Consumer。其運行模式大致為下圖所示。
Sequence Diagram



可大致理解為由Consumer先向Broker訂閱某個topic。當任一Producer向這個topic發布消息的時候，所有訂閱此topic的Consumer都能收到此消息。

Kafka Producer’s view

以Producer的觀點來分析Kafka，著重在Producer的核心功能
send

```
Future<RecordMetadata> send(ProducerRecord<K, V> record, Callback callback);
```

- 當send被call時，Producer會先向Cluster發送API version與Kafka metadata的請求。做protocol的確認與獲得主要負責該topic,partition 的Broker的資訊。取得的資料會被存進 ProducerMeta，方便之後的send可以直接由此處調用所需資料。
- 接著要發送的訊息會被暫存到RecordAccumulator，當暫存的record總大小超過ProducerConfig所設定的BATCH_SIZE，或是經過設定的LINGER_MS時長後，會把RecordAccumulator中的records包裝成一個Produce Request發送到Broker。Broker在處裡好訊息後會回傳一個Producer Response，讓Producer可在CallBack中對error做處裡。

若是想立即發送訊息也可以使用 flush() 來將當前Accumulator中的訊息發出

PARTITION

Kafka有類似於Mqtt的sharding的概念，叫做partition。可以把一個topic切成多段處裡，使系統可以平行擴充，架設多個Broker來分擔壓力。

如何決定一個訊息該發到哪個partition?

Producer可以在send的時候"選擇"加入partition number 以及 key 的參數。若有設定partition number，則該訊息就會被發到對應編號的partition。若沒有partiton number，則由key的hash決定。若只單純發送value的話，一個producer就會流輪發送到不同partition。

Kafka Consumer’s view

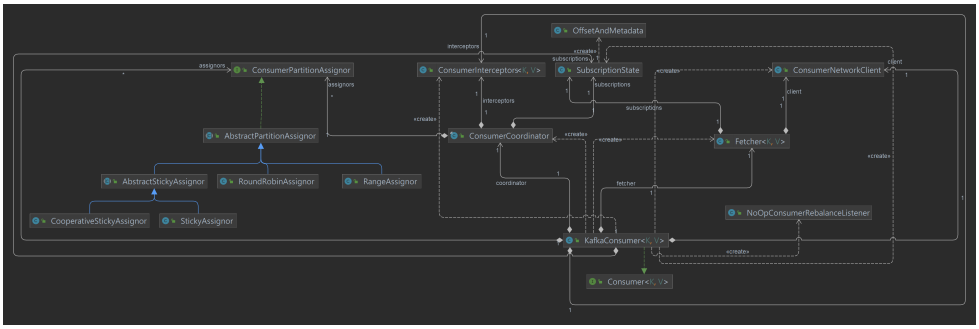
以Consumer的觀點來分析Kafka，著重在Consumer核心功能Subscribe(), Poll()以及Kafka透過Rebalance來進行有效率的資源分配。會先透過Class Diagram對於Kafka Consumer整體架構有初步的了解，整體的類別圖會著重在KafkaConsumer與核心功能Class間的繼承、實作介面、呼叫，能有助於對Kafka Consumer的理解以及往後的實作細節。Rebalance功能會使用Sequence Diagram時序圖來表示，因為觸發Rebalance事件需要類似Topoic中Partitions數

量改變，Consumer Group中有新成員加入或退出，這樣的事件都是涉及時間及元件之間的互動，因此會用較直觀的時序圖來表達這樣的關係。

Class Diagram

Class Diagram透過IntelliJ IDEA內建的UML所繪製。下圖所呈現的Class Diagram並非是與Consumer所有相關聯的功能都繪製在此圖。此圖著重在實作Consumer介面的KafkaConsumer中核心功能。

- ConsumerCoordinator : 繼承 AbstractCoordinator介面，管理與consumer coordinator相關的功能例如Group Registration, Group/Leader Selection, State Assignment功能。且透過此才能與ConsumerGroupMetadata溝通。
- ConsumerPartitionAssignor : 用來定義Consumer partitions如何分配的介面，Kafka四種分配策略也是繼承此介面分別是RangeAssignor, RoundRobinAssignor, StickyAssignor, CooperativeStickyAssignor。
- ConsumerNetworkClient : 提供Consumer與網路接取的相關功能
- Fetcher : 負責向Broker獲取訊息的相關功能且是Thread-safety
- NoOpConsumerRebalanceListener : 實作ConsumerRebalanceListener callback interface，當觸發Rebalance時所採取的行動，前提是Consumer採用Kafka auto-manage group membership才適用。
- SubscriptionState : 用來追蹤紀錄Consumer的topics, partitions, and offsets
- OffsetAndMetadata : 當offest被確認後允許使用者提交額外的元資料(哪個節點哪個時間)。
- ConsumerInterceptors<K, V>：存有list傳入custom interceptors，可以在poll(),offest傳回成功與否前進行攔截及修改。



Subscribe細節解析

Kafka Scbscribe Source code

(https://github.com/apache/kafka/blob/7c8c65fc54b9eb0386787632ff8315e097d99818/clients/src/main/java/org/apache/kafka/clients/consumer/KafkaConsumer.java#L956-L980)

```java
@Override
public void subscribe(Collection<String> topics, ConsumerRebalanceListener listener) {
    acquireAndEnsureOpen();
    try {
        maybeThrowInvalidGroupIdException();
        if (topics == null)
            throw new IllegalArgumentException("Topic collection to subscribe to cannot be null");
        if (topics.isEmpty()) {
            // treat subscribing to empty topic list as the same as unsubscribing
            this.unsubscribe();
        } else {
            for (String topic : topics) {
                if (Utils.isBlank(topic))
                    throw new IllegalArgumentException("Topic collection to subscribe to cannot contain null or empty topic");
            }

            throwIfNoAssignorsConfigured();
            fetcher.clearBufferedDataForUnassignedTopics(topics);
            log.info("Subscribed to topic(s): {}", Utils.join(topics, ", "));
            if (this.subscriptions.subscribe(new HashSet<>(topics), listener))
                metadata.requestUpdateForNewTopics();
        }
    } finally {
        release();
    }
}
```

在這之前會先有初始化設置配置，傳入的參數會是Key Value。包括Heartbeat頻率，是否要自動更新Offset，Key Value的反序列化，Timer時間等等。

1. 先確定Consumer在不在且取得lock
    - `acquireAndEnsureOpen();`
2. 檢查GroupID存不存在
    - `maybeThrowInvalidGroupIdException();`
3. 訂閱的Topic不能是empty or null
    - `if (Utils.isBlank(topic))`
4. 檢查有無Partition assigner
    - `throwIfNoAssignorsConfigured();`
    - `private List<ConsumerPartitionAssignor> assignors;`

- 裡面會去檢查 `if (assignors.isEmpty())`
- 如果是空的會 `throw new IllegalStateException`

5. 實際訂閱動作
   - `fetcher.clearBufferedDataForUnassignedTopics(topics);`
   - `Set<TopicPartition> currentTopicPartitions = new HashSet<>();`
   - TopicPartition在kafka/common/TopicIdPartition.java
   - 把新訂閱的加到 `currentTopicPartitions.add()`
   - 把非此次的訂閱的Topic從buffer中清掉
6. 訂閱的Topic(s) 寫入log
   - `log.info()`
7. 更新時戳
   - `if (this.subscriptions.subscribe(new HashSet<>(topics), listener))`
   - `requestUpdateForNewTopics()`
   - Override the timestamp of last refresh to let immediate update
8. 釋放掉lock
   - `release()`
   - Release the light lock protecting the consumer from multi-threaded access.

Poll 細節解析

Kafka Poll Source code

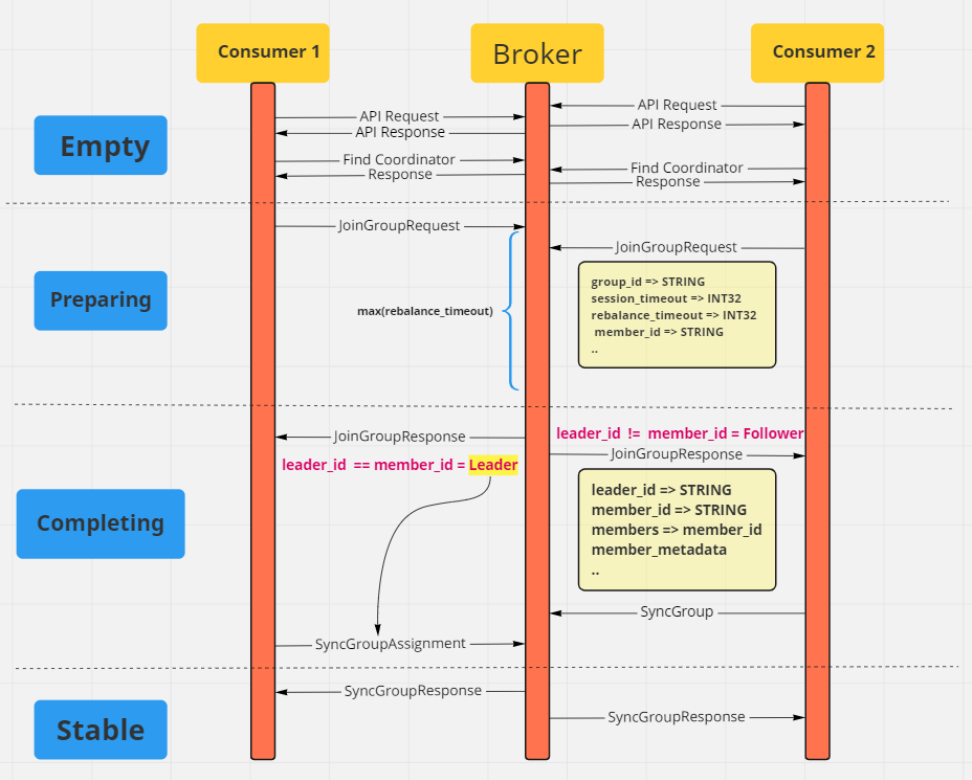
(https://github.com/apache/kafka/blob/7c8c65fc54b9eb0386787632ff8315e097d99818/clients/src/main/java/org/apache/kafka/clients/consumer/KafkaConsumer.java#L1226-L1273)

```java
private ConsumerRecords<K, V> poll(final Timer timer, final boolean includeMetadataInTimeout) {
    acquireAndEnsureOpen();
    try {
        this.kafkaConsumerMetrics.recordPollStart(timer.currentTimeMs());

        if (this.subscriptions.hasNoSubscriptionOrUserAssignment()) {
            throw new IllegalStateException("Consumer is not subscribed to any topics or assigned any partitions");
        }

        do {
            client.maybeTriggerWakeup();

            if (includeMetadataInTimeout) {
                // try to update assignment metadata BUT do not need to block on the timer for join group
                updateAssignmentMetadataIfNeeded(timer, false);
            } else {
                while (!updateAssignmentMetadataIfNeeded(time.timer(Long.MAX_VALUE), true)) {
                    log.warn("Still waiting for metadata");
                }
            }
             final Fetch<K, V> fetch = pollForFetches(timer);
             if (!fetch.isEmpty()) {
                // before returning the fetched records, we can send off the next round of fetches
                // and avoid block waiting for their responses to enable pipelining while the user
                // is handling the fetched records.
                //
                // NOTE: since the consumed position has already been updated, we must not allow
                // wakeups or any other errors to be triggered prior to returning the fetched records.
                if (fetcher.sendFetches() > 0 || client.hasPendingRequests()) {
                    client.transmitSends();
                }

                if (fetch.records().isEmpty()) {
                    log.trace("Returning empty records from `poll()` "
                            + "since the consumer's position has advanced for at least one topic partition");
                }

                return this.interceptors.onConsume(new ConsumerRecords<>(fetch.records()));
            }
        } while (timer.notExpired());

        return ConsumerRecords.empty();
    } finally {
        release();
        this.kafkaConsumerMetrics.recordPollEnd(timer.currentTimeMs());
    }
}
```

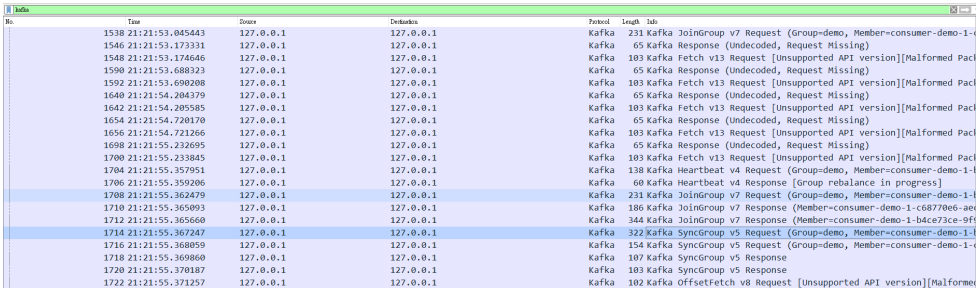
1. 先確定Consumer在不在且取得lock
    - `acquireAndEnsureOpen();`
2. 紀錄Consumer poll的行為
    - 紀錄此次的poll時間及距離上次poll的時間
    - `this.kafkaConsumerMetrics.recordPollStart(timer.currentTimeMs());`
3. 檢查Consumer有沒有訂閱主題
    - `hasNoSubscriptionOrUserAssignment()`
    - 如果沒有會丟出 `IllegalStateException()`
4. 每個Poll都要先確保對外連接的client是否被喚醒
    - `client.maybeTriggerWakeup();`
5. 嘗試更新分配的metadata且不會影響到Join Group的Timer
    - default是true `includeMetadataInTimeout`
    - `updateAssignmentMetadataIfNeeded(timer, false);`

6. 先確保fetch到的record數量不是0
7. 因為此次回傳的值都是上次Fetch到的內容
    - 此段在做檢查並讓效率提高作Pipeline
    - 符合的話會在此次送出回傳給下一次的Fetch
8. 如果fetch到的records是empty寫入log
    - 這可能是因為某些因素被中止而導致的
9. 回傳fetch到的records
    - Default每次最大fetch的record數量500
10. 重複5-10直到Timer過期
11. 回傳一個空的Key Value值
12. 釋放掉lock
    - `release()`
13. 將此次的Poll的衡量指標記錄到kafkaConsumerMetrics

Rebalance Sequence Diagram



Rebalance機制運用兩個不同的Consumer加到同一個Consumer Group的情況來解釋且有用Wireshark Trace過程再將此過程透過Miro工具繪製。



No.	Time	Source	Destination	Protocol	Length	Info
1538	21:21:53.045443	127.0.0.1	127.0.0.1	Kafka	231	Kafka JoinGroup v7 Request (Group=demo, Member=consumer-demo-1-c
1546	21:21:53.173331	127.0.0.1	127.0.0.1	Kafka	65	Kafka Response (Undecoded, Request Missing)
1548	21:21:53.174646	127.0.0.1	127.0.0.1	Kafka	103	Kafka Fetch v13 Request [Unsupported API version][Malformed Pacl
1590	21:21:53.688323	127.0.0.1	127.0.0.1	Kafka	65	Kafka Response (Undecoded, Request Missing)
1592	21:21:53.690208	127.0.0.1	127.0.0.1	Kafka	103	Kafka Fetch v13 Request [Unsupported API version][Malformed Pacl
1640	21:21:54.204379	127.0.0.1	127.0.0.1	Kafka	65	Kafka Response (Undecoded, Request Missing)
1642	21:21:54.205585	127.0.0.1	127.0.0.1	Kafka	103	Kafka Fetch v13 Request [Unsupported API version][Malformed Pacl
1654	21:21:54.720170	127.0.0.1	127.0.0.1	Kafka	65	Kafka Response (Undecoded, Request Missing)
1656	21:21:54.721266	127.0.0.1	127.0.0.1	Kafka	103	Kafka Fetch v13 Request [Unsupported API version][Malformed Pacl
1698	21:21:55.232695	127.0.0.1	127.0.0.1	Kafka	65	Kafka Response (Undecoded, Request Missing)
1700	21:21:55.233845	127.0.0.1	127.0.0.1	Kafka	103	Kafka Fetch v13 Request [Unsupported API version][Malformed Pacl
1704	21:21:55.357951	127.0.0.1	127.0.0.1	Kafka	138	Kafka Heartbeat v4 Request (Group=demo, Member=consumer-demo-1-b
1706	21:21:55.359206	127.0.0.1	127.0.0.1	Kafka	60	Kafka Heartbeat v4 Response [Group rebalance in progress]
1708	21:21:55.362479	127.0.0.1	127.0.0.1	Kafka	231	Kafka JoinGroup v7 Request (Group=demo, Member=consumer-demo-1-b
1710	21:21:55.365093	127.0.0.1	127.0.0.1	Kafka	186	Kafka JoinGroup v7 Response (Member=consumer-demo-1-c68770e6-ae
1712	21:21:55.365660	127.0.0.1	127.0.0.1	Kafka	344	Kafka JoinGroup v7 Response (Member=consumer-demo-1-b4ce73ce-9f
1714	21:21:55.367247	127.0.0.1	127.0.0.1	Kafka	322	Kafka SyncGroup v5 Request (Group=demo, Member=consumer-demo-1-b
1716	21:21:55.368059	127.0.0.1	127.0.0.1	Kafka	154	Kafka SyncGroup v5 Request (Group=demo, Member=consumer-demo-1-c
1718	21:21:55.369860	127.0.0.1	127.0.0.1	Kafka	107	Kafka SyncGroup v5 Response
1720	21:21:55.370187	127.0.0.1	127.0.0.1	Kafka	103	Kafka SyncGroup v5 Response
1722	21:21:55.371257	127.0.0.1	127.0.0.1	Kafka	102	Kafka OffsetFetch v8 Request [Unsupported API version][Malformed

Consumer Group狀態為Empty

1. Consumer會先向Broker要後續溝通所需的API及版本
2. Consumer會收到Broker所回應的API版本
3. Consumer會去找一個特定Broker作為coordinator負責監控再加入Consumer Group後的相關事項(Hearbeat,啟動Rebalance等)
4. Consumer會收到回應的特定coordinator

Consumer Group狀態為Preparing Balance

5. Consumer會去申請加入ConsumerGroup
    - group id : 要加入的group id
    - session_timeout : 不用heartbeat也能確保Consumer還在
    - rebalance_timeout : 用來等Consumer重新加入群組的時間

因此Broker在最大的rebalance_timeout時間後就會把Response送出

Consumer Group狀態為Completing Balance

6. Consumer會收到Broker的JoinGroup回應
    - 通常第一個送要JoinGroupRequst的消費者會是Leader
    - 可以由leader_id及member_id去判斷為Leader或Follower
    - 內有各組員訂閱狀況
7. Follower會送出SyncGroup給Broker
    - 但內容為空意思在等待leader分配後的結果
8. Leader會收集JoinGroup回應後做出分配並傳給Broker
    - 分配策略就涉及到上面所提ConsumerPartitionAssignor介面

- RangeAssignor, RoundRobinAssignor, StickyAssignor, CooperativeStickyAssignor

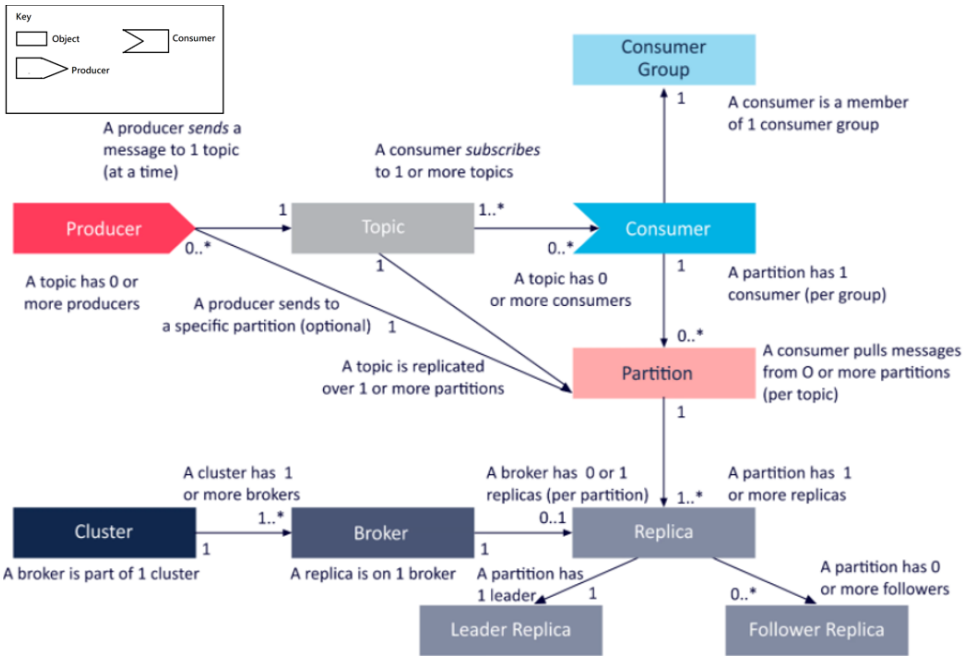
Consumer Group狀態為Stable

10. 最後每位Consumer會收到被分配的狀況

直到下次觸發Rebalance事件發生Stable狀態才會改變

Kafka Zookeeper Broker's view

- 動機：Zookeeper 做好了分散式系統的協調服務讓分散式系統直接引用，免去從頭開始實作協調服務的麻煩。
- Reliability: Zookeeper 可以在有些節點fail 時，依然持續運作。
- Simplicity:Zookeeper 可以藉由shared hierarchical namespace 協調process。
- Speed: Zookeeper 可以快速的處理Reading的workloads。
- Scalability:Zookeeper是horizontally scalable。可以藉由僅僅增加一個node來擴充。

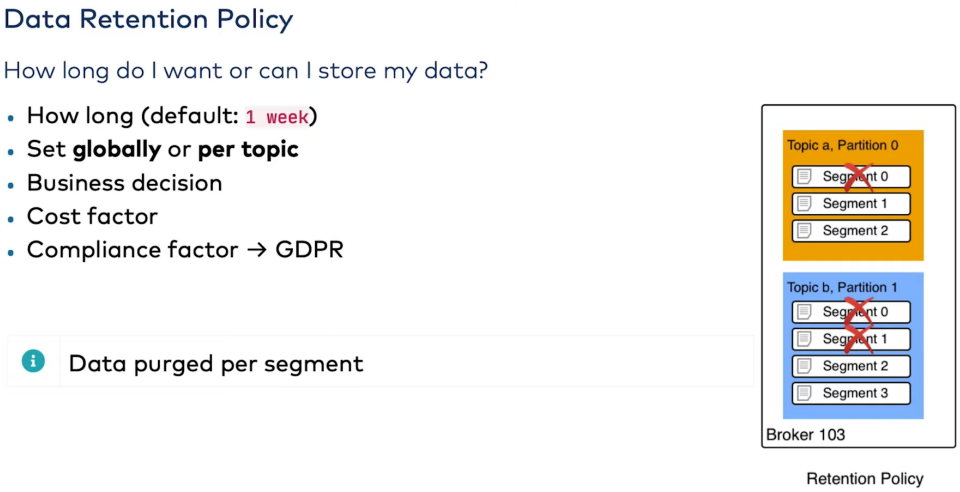


Broker 在Zookeeper中的註冊

- 為了記錄Broker的註冊訊息，在Zookeeper上專門創建了屬於Kafka的一個節點，其路徑為/brokers
- Kafka的每個Broker啟動時，都會到Zookeeper中進行註冊，告訴Zookeeper其 **broker.id** (http://broker.id)。而在整個群集中，broker.id全局唯一，並在Zookeeper中創建其屬於

自己的節點，其節點路徑為/brokers/ids/{**broker.id** (http://broker.id)}。

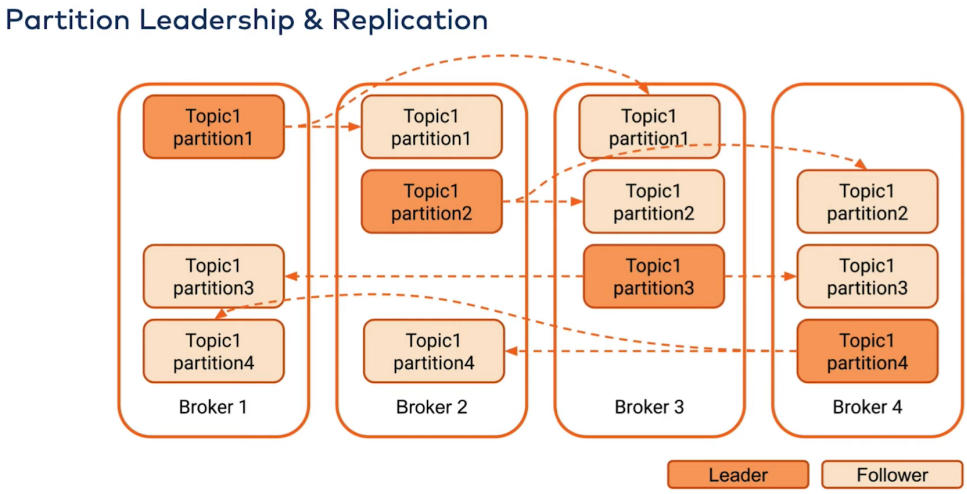
- 創建完節點後，Kafka會將該broker的broker.name及其port記錄到該節點。
- 另外，該broker節點屬性為臨時節點，當broker失效時，Zookeeper會刪除該節點，這樣我們就可以很方便的監控到broker節點的變化及rebalance。
- 我們可以設定broker中的data要儲存多久，因為維持越久的data，會花費越多的成本來維護。

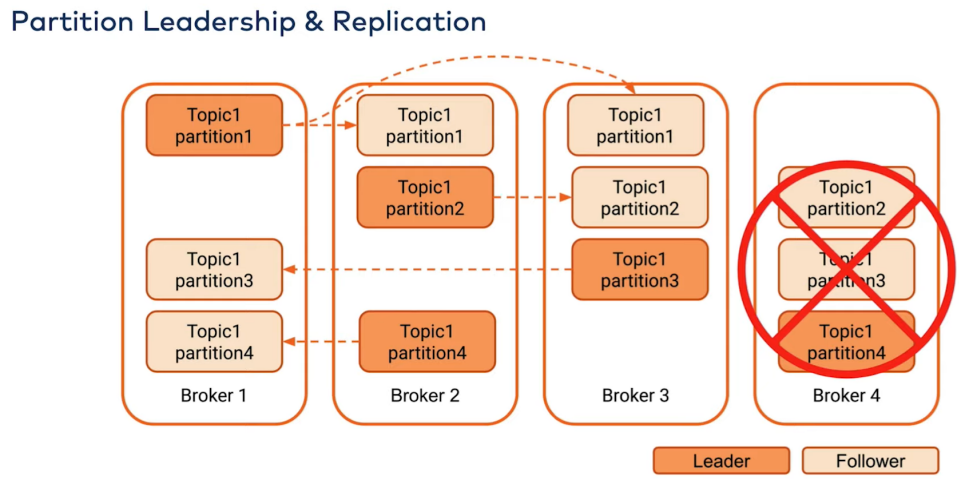


Retention Policy

Topic 在Zookeeper中的註冊

- 在Kafka中，所有topic及broker的對應關係都由Zookeeper進行維護。在Zookeeper中，建立專門的節點來記錄這些訊息，其節點路為/brokers/topics/{topic_name}。
- 為了保障數據的可靠性，每個Topic的partitions 實際上是存在備份的，並且備份數量由Kafka機制中的replicas來控制。為了保障數據的一致性，Kafka為每一個partition找一個節點作為leader，其餘備份作為follower。當producer push 的訊息寫入partition時，作為leader的broker會將訊息寫入自己的分區，同時還會將訊息複製到各個follower，實現同步。而follower只是跟隨leader的變動去更新資料。如果某個follower掛掉，leader會再找一個替代，並同步訊息。如果leader掛了，follower們會選舉出一個新的leader代替，繼續服務，這些都是由Zookeeper完成的。





Consumer 在Zookeeper中的註冊

當新的consumer 註冊到Zookeeper時，Zookeeper會創建專用的節點來保存相關的訊息，其節點路徑為ls/consumers/{group_id}。其節點下有三個子節點，分別為[ids, owners, offsets]。

- ids節點:紀錄該消費組中正在消費的consmer。
- owners節點:紀錄該消費組消費的topic訊息。
- offsets節點:紀錄每個topic的每個分區的offset。

借助Zookeeper實現Rebalance Sequence

Consumer在消費時只需制定topic，借助Zookeeper可以根據partition的數量和consumer的數量做到均衡的動態配置。Consumer在啟動時會到Zookeeper下以自己的conusmer-id創建臨時節點/consumer/[group-id]/ids/[conusmer-id]， 並對/consumer/[group-id]/ids 註冊監聽事件。當consumer發生變化時，同一個group的其他consumer會得到通知。

紀錄Consumer進度的offset

在consumer對指定的訊息進行消費的過程中，需要定時的將partition消息的消費進度offset 記錄到Zookeeper中，以便在該consumer進行重啟或者其他consumer重新接管該消息

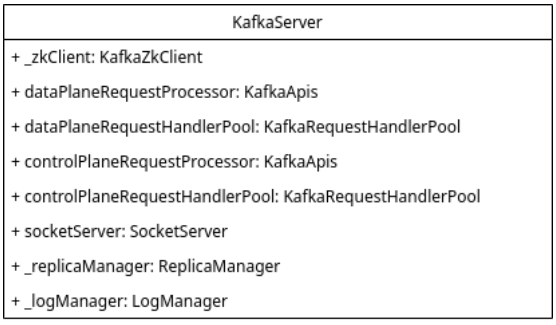
partition的消費權後，能夠從之前的進度開始繼續進行消費。Offset 在Zookeeper中由一個專門的節點進行紀錄，其節點路為
/consumers/[group_id]/offsets/[topic]/[broker_id-partition_id]

紀錄consumer與partition的關係

在Kafka中，規定了每個partition只能被同組的一個消費者進行消費。因此，需要在Zookeeper上記錄下partition與consumer之間的關係。每個consumer一旦確定了對一個partition的消費權利，需要將其consumer ID寫入到Zookeeper對應消息分區的臨時節點上。例如:
/consumers/[group_id]/owners/[topic]/[broker_id-partition_id]
其中，[broker_id-partition_id] 就是一個消息分區的標示，節點內容就是該消息分區consumer的consumer ID。

BROKER 啟動流程

broker 是從路徑 `core/src/main/scala/kafka/Kafka.scala` 開始執行，會以讀入的 properties 檔來啟動不同的 broker，如果以 `config/server.properties` 的範例設定來設定，會採用 Zookeeper 來同步 replica node ，若以 `config/kraft/server.properties` 的範例設定設定時，則會以 Raft 的方式來同步。以 Zookeeper 來設定則會產生 `KafkaServer` 的物件， `KafkaServer` 的主要成員如下圖



`_zkClient` 為 zookeeper server 的 client 端，會對其他相同連線到 zookeeper server 的 replica node 發送同步， `dataPlaneRequestProcessor` 、 `dataPlaneRequestHandlerPool` 、 `controlPlaneRequestProcessor` 、 `controlPlaneRequestHandlerPool` 、 `socketServer` 則是處理從網路來的請求，一開始會將 RequestProcessor 以及 RequestHandlerPool 註冊進 `sockeServer` 的 data plane 以及 control plane，當有新的 Request 時， `socketServer` 便會從 RequestHandlerPool 產生對應的 RequestHandler 並執

行， `_replicaManager` 是管理節點的 partition，依照 partition 的同步狀況來決定是否繼續同步該 partition，partition 內容紀錄則是靠 `_logManager` 來完成，`_logManager` 會根據設定檔中的 **log.dirs** 的路徑存放 consumer 、topic 、等與 broker 相關的資料。以預設的 `config/server.properties` 來說，會存放在 **/tmp/kafka-logs**。

技術實作與心得

Demo Source Code

(https://github.com/SpeedReach/KafkaTesting/tree/main/src/main/java/team/antipython/kafkatest)

首先把Cluster架起來，開啟兩個terminal，分別運行 Zookeeper 與 1個Kafka Broker。
在server.properties中設定 broker的 ip與host，
這邊設定 `localhost:9092`

```
.\zookeeper-server-start.bat .\zookeeper.properties
.\kafka-server-start.bat .\server.properties
```

Cluster架好後來要創建Topics，這邊創建一個名叫 DemoTopic的Topic

```
Admin admin = Admin.create(Collections.singletonMap(AdminClientConfig.BOOTSTRAP_SERVERS_
try {
    Set<String> existingTopics = admin.listTopics().names().get();
    if(!existingTopics.contains("DemoTopic")){
        /*
        Partitions 是該topic的訊息可以分成幾類
        ReplicationFactors 是副本數 必須小於 Broker數量
        由於demo只有一個 broker所以只有一個副本
        */
        int partitions = 3;
        short replicationFactors = 1;
        Set<NewTopic> newTopics = Collections.singleton(new NewTopic("DemoTopic",partiti
        admin.createTopics(newTopics).all().get();
    }
} catch (ExecutionException | InterruptedException e) {
            e.printStackTrace();
}
```

接著寫Consumer來訂閱DemoTopic的訊息

```
    Properties props = new Properties();
    props.setProperty("bootstrap.servers", "localhost:9092");
    props.setProperty("group.id", "demo");
    props.setProperty("enable.auto.commit", "false");
    props.setProperty("key.deserializer", "org.apache.kafka.common.serialization.StringD
    props.setProperty("value.deserializer", "org.apache.kafka.common.serialization.Strin
    final Consumer<String,String> consumer = new KafkaConsumer<>(props);
    consumer.subscribe(Collections.singleton("DemoTopic"));
    while (true){
        ConsumerRecords<String,String> records = consumer.poll(Duration.ofMillis(100));
        records.forEach(record->{
            System.out.println("Topic: "+record.topic() + " ,Value: "+record.value());
        });
    }
```

接著來寫Producer發布一條"SomeMessage"的訊息到DemoTopic

```
Properties props = new Properties();
props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
props.put(ProducerConfig.ACKS_CONFIG, "all");
props.put(ProducerConfig.RETRIES_CONFIG, 0);
props.put("linger.ms", 1);
Producer<String, String> producer = new KafkaProducer<>(props);

producer.send(new ProducerRecord<>("DemoTopic","SomeMessage"),(recordMeta,err)->{
    System.out.println("Sent message to "+recordMeta.topic)
});
producer.flush();
```

先執行**Admin**(https://github.com/SpeedReach/KafkaTesting/blob/main/src/main/java/team/antipython/kafkatest/Init.java#L25-L59)創建Topic後，再執行**Producer**(https://github.com/SpeedReach/KafkaTesting/blob/main/src/main/java/team/antipython/kafkatest/Publisher.java#L22-L39)與**Consumer**(https://github.com/SpeedReach/KafkaTesting/blob/main/src/main/java/team/antipython/kafkatest/Subscriber.java#L22-L42)，可看到Consumer輸出

```
Topic: DemoTopic ,Value: SomeMessage
```

Producer則輸出

```
Sent message to DemoTopic
```

評論

- Identify a context where this technology is not appropriate and justify your answer
  - 我需要一個分散式系統，在晚上時Publisher發出「晚上了」給Subscriber，早上時發出「早上了」，且Subscriber數量不超過100個，此時使用Kafka就顯得大材小用了。
  - 當每筆data都需要保存至少10年以上，因為Kafka需要複製儲存副本，此時使用Kafka就需要龐大的儲存成本。
- Identify a context where this technology is appropriate and justify your answer
  - 今天我需要在將data傳給Consumer之前先進行一次處理，此時就可以使用Kafka，將Producer傳進來的data先給data handlers處理後再儲存。

結語

Kafka設計的初衷在於處理即時資料提供一個統一、高吞吐、低延遲的平台，因此設計的重點更著重在效能與擴充性，但也包括了MQ本來就具有的優點，例如:不同步的溝通、可靠性等等，底下分別提到這些特性。
負載平衡:透過partition的設計，producer發布訊息在不同broker上，以partition為單位，可以依據partition目錄的訊息量、broker目前的負載做負載平衡。
減少耦合:透過topic的設計，producer、consumer只要關心borker在哪，以及所選的topic即可，不須知道對方的資訊。
順序保證:在partition中，又切割成segment為儲存單位，收到的訊息皆以循序的方式儲存，可以確保partition內部的順序儲存；對於consumer來說，由於一個訊息只能有一個consumer消耗，因此當consumer數量大於等於partition時，還可以再確保收到的訊息順序。
不同步溝通:kafka設計中允許In-Sync Replicas以外的副本不必同步更新，透過設定min.insync.replicas參數以及In-Sync Replicas數目，在可靠度與效率之間可以有彈性的調整。
擴展性:可依照需求增加或減少partition。
可恢復性:在partition中儲存的訊息中，透過HW機制以及broker定時向zookeeper更新參數，當某些broker故障時，因為有副本存在，並不影響其他節點運作，而在系統恢復之後，可以利用副本以及HW恢復遺失的資料。
緩衝:利用message queue做為緩衝，consumer採用pull的方式獲取資料，可以照顧到不同效率的consumer設備，使producer、consumer不必互相等待。
高速存取:直接存取磁碟而不必先複製到記憶體，且存取代價為O(1)。
Kafka適合應用在大部分大規模服務的情況，且consumer可以以pull方式接收訊息；反之，若只是應用在一般的整合服務(例如企業應用整合)，並不會有太多節點，訊息量可能也不大，只是為了整合不同系統，那使用kafka就顯得小題大作，浪費了可擴充性、大吞吐量等特性。

分工

- 問題(目的)與背景: 仁傑
- 架構解析 :
    - Broker : 庭慶 冠宏
    - Consumer : 皓博

    - Producer : 詹霖
- 技術實作與心得 : 詹霖
- 評論 : 仁傑
- 結語 : 盛泰
- 書面資料