消息队列 Pulsar 版

开发指南

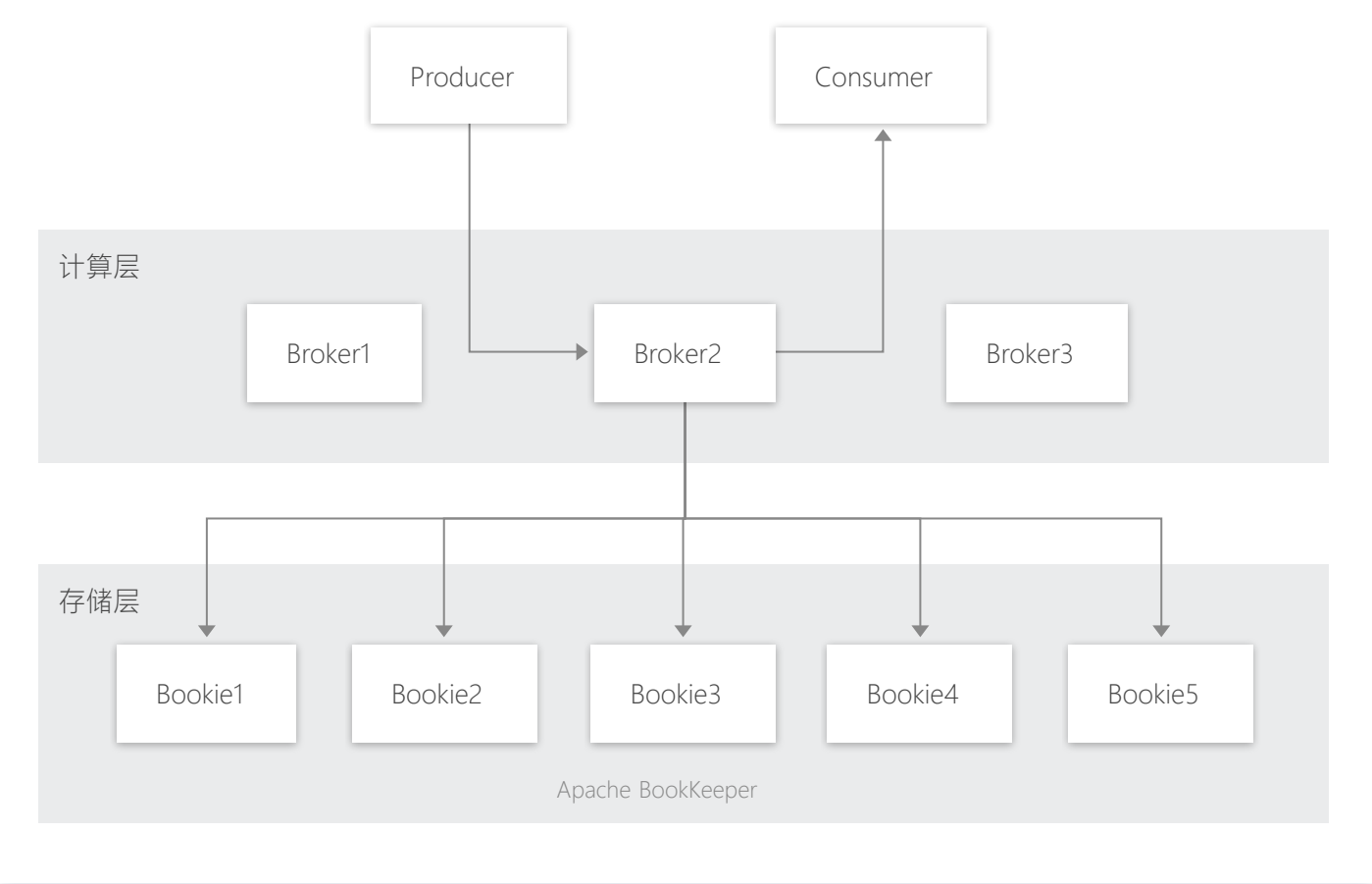
文档目录

开发指南
原理解析
Pulsar Topic 和分区

最近更新时间：2021-07-02 15:09:52

Apache Pulsar 架构

Apache Pulsar 是一个发布-订阅模型的消息系统，由 Broker、Apache BookKeeper、Producer、Consumer 等组件组成。



- Producer： 消息的生产者，负责发布消息到 Topic。
- Consumer：消息的消费者，负责从 Topic 订阅消息。
- Broker：无状态服务层，负责接收和传递消息，集群负载均衡等工作，Broker 不会持久化保存元数据，因此可以快速的上、下线。
- Apache BookKeeper：有状态持久层，由一组 Bookie 存储节点组成，可以持久化地存储消息。

Apache Pulsar 在架构设计上采用了计算与存储分离的模式，消息发布和订阅相关的计算逻辑在 Broker 中完成，数据存储在 Apache BookKeeper 集群的 Bookie 节点上。

Topic 与分区

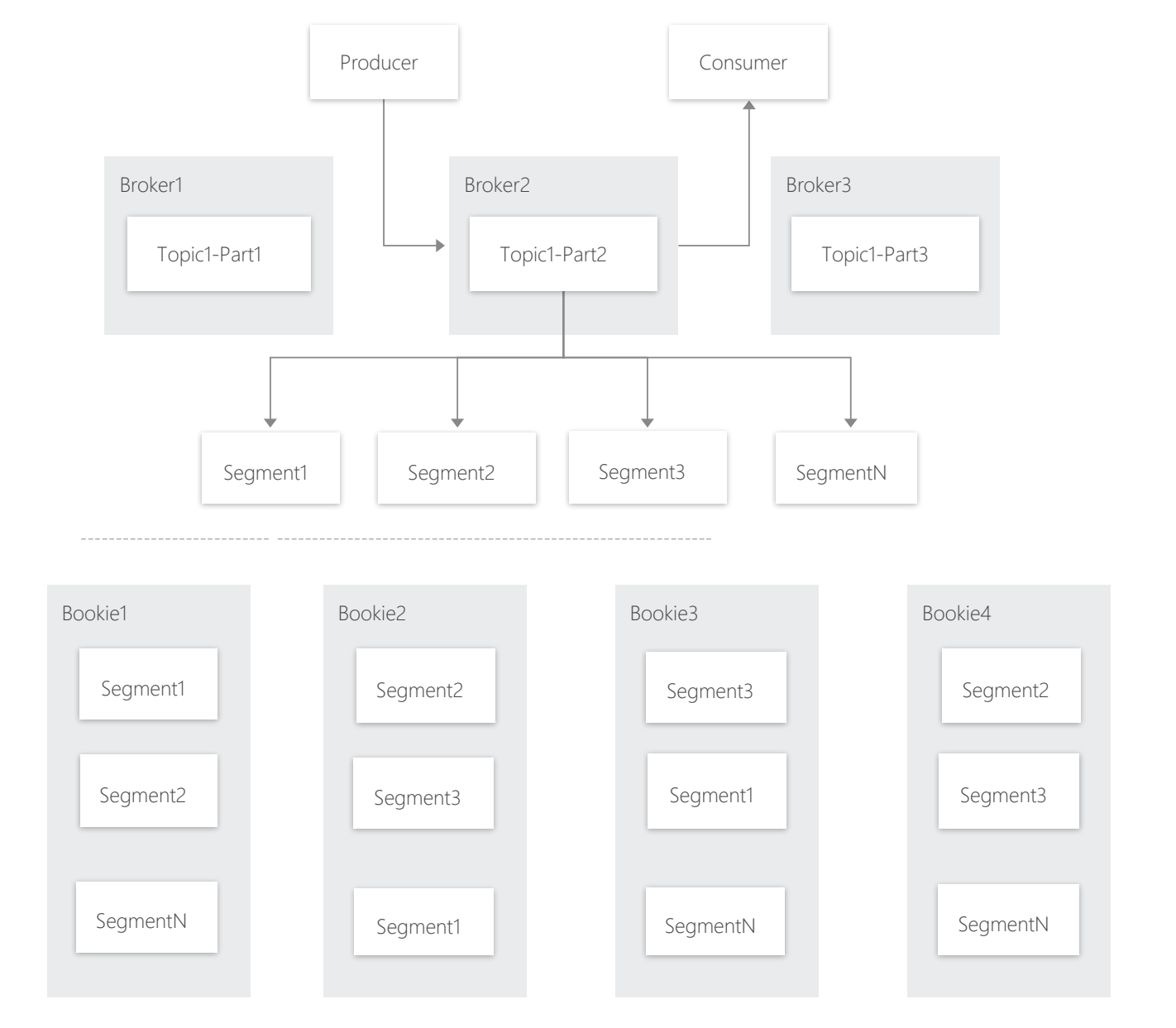
Topic（主题）是某一种分类的名字，消息在 Topic 中可以被存储和发布。生产者往 Topic 中写消息，消费者从 Topic 中读消息。

Pulsar 的 Topic 分为 Partitioned Topic 和 Non-Partitioned Topic 两类，Non-Partitioned Topic 可以理解为一个分区数为1的 Topic。实际上在 Pulsar 中，Topic 是一个虚拟的概念，创建一个3分区的 Topic，实际上是创建了3个“分区Topic”，发给这个 Topic 的消息会被发往这个 Topic 对应的多个 “分区Topic”。

例如：生产者发送消息给一个分区数为3，名为 `my-topic` 的 Topic，在数据流向上是均匀或者按一定规则（如果指定了key）发送给了 `my-topic-partition-0` 、 `my-topic-partition-1` 和 `my-topic-partition-2` 三个“分区Topic”。

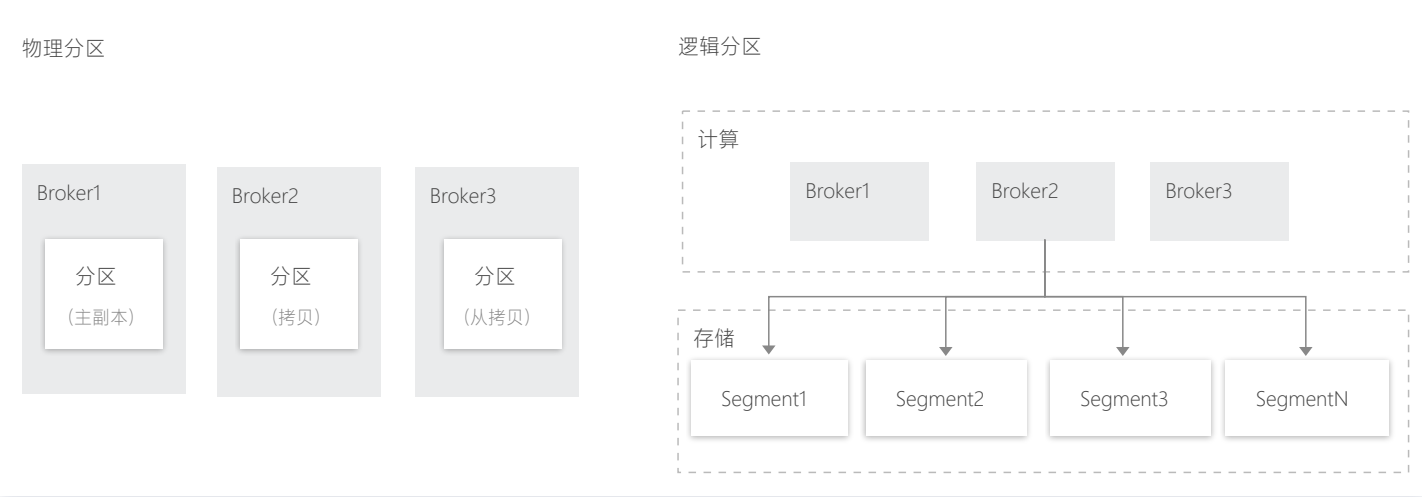
分区 Topic 做数据持久化时，分区是逻辑上的概念，实际存储的单位是分片（Segment）的。

如下图所示，分区 Topic1-Part2 的数据由N个 Segment 组成， 每个 Segment 均匀分布并存储在 Apache BookKeeper 群集中的多个 Bookie 节点中， 每个 Segment 具有3个副本。



物理分区与逻辑分区

逻辑分区和物理分区对比如下：



物理分区：计算与存储耦合，容错需要拷贝物理分区，扩容需要迁移物理分区来达到负载均衡。

逻辑分区：物理“分片”，计算层与存储层隔离，这种结构使得 Apache Pulsar 具备以下优点。

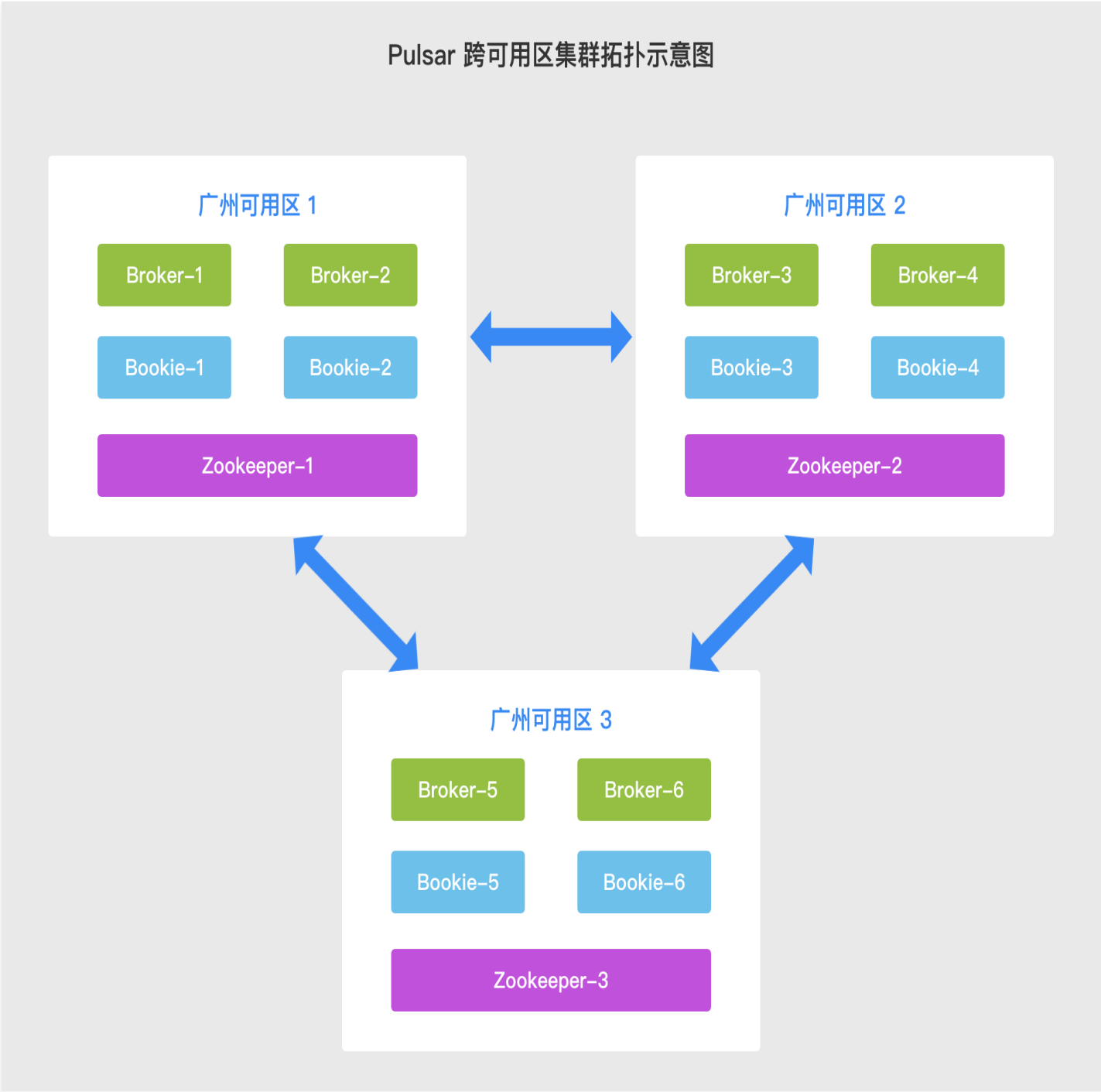
- Broker 和 Bookie 相互独立，方便实现独立的扩展以及独立的容错。
- Broker 无状态，便于快速上、下线，更加适合于云原生场景。
- 分区存储不受限于单个节点存储容量。
- 分区数据分布均匀，单个分区数据量突出不会使整个集群出现木桶效应。
- 存储不足扩容时，能迅速利用新增节点平摊存储负载。

Pulsar 跨可用区部署

最近更新时间：2023-07-11 10:57:21

Pulsar 跨可用区部署

Pulsar 支持跨可用区部署，即在有 3 个或 3 个以上可用区的地域购买 Pulsar 集群时，可以最多选择 3 个可用区购买跨可用区实例。该实例分区副本会强制分布在各个可用区节点上，这种部署方式能够让您的实例在单个可用区不可用情况下仍能正常提供服务。



跨可用区部署原理

Pulsar 跨可用区容灾是通过机架感知原理实现，即将不同组件的服务节点部署到不同可用区的不同机架中，但本质还是一套 Pulsar 集群。

机架感知是 Ensemble Placement Policy 的一种，是 Bookkeeper Client 用来选择 Ensembles 的算法，选择的依据主要是依赖网络拓扑属性。

如何选择 Bookie

在初始化部署 Bookie 集群时，Bookie 会向当前的 Zookeeper 注册一个临时的 zk-node，Bookkeeper Client 通过 Zookeeper（watcher）来发现 Bookie 列表。当 bookie 变更时（宕机），Ensemble Placement Policy 会通过 onClusterChanged(Set, Set) 接口被通知，然后重新构造新的网络拓扑，后续的操作，例如 newEnsemble 就会基于新的网络拓扑来生成。

NetworkTopology

网络拓扑用一个树状的分层结构来表示一个集群中的 Bookie 节点信息。一个 Bookie 集群可以由很多的 data center（region）组成。在一个 data center 内部包含了分布在不同 rack 上的机器，在树状结构中，叶子节点表示 bookie 信息。

- **示例1：** Region A 有三个 bookie， bk1 , bk2 and bk3 , 它们的网络位置是， /region-a/rack-1/bk1 , /region-a/rack-1/bk2 , /region-a/rack-2/bk3 , 网络拓扑结构就是如下所示：

```
     root
      |
   region-a
    / \
rack-1  rack-2
 / \     \
 bk1  bk2    bk3
```

- **示例2：** Region A 和 Region B 有四个 bookie， bk1 , bk2 , bk3 and bk4 ，它们的网络位置是，/region-a/rack-1/bk1 , /region-a/rack-1/bk2 , /region-b/rack-2/bk3 和 /region-b/rack-2/bk4 ，网络拓扑结构如下：

```
        root
       /    \
   region-a     region-b
     |           |
  rack-1        rack-2
   / \         /   \
 bk1  bk2      bk3   bk4
```

网络位置是通过 DNSToSwitchMapping 来解析的， DNSToSwitchMapping 会将域名或者 IP 解析到网络位置。网络位置一定是 /region/rack 格式， / 表示 root，region 表示 data center，rack 表示机架信息。

> **说明**

> 在 Pulsar 中实现的 DNSToSwitchMapping 是 BookieRackAffinityMapping，通过 ZK 保存 rack 信息，目前只实现了 rackaware 的能力。

使用说明

目前仅专业集群支持跨可用区部署。请参考 产品选型 。

> **说明**
> 目前该功能公测中，暂不单独计费。

消息存储原理与 ID 规则

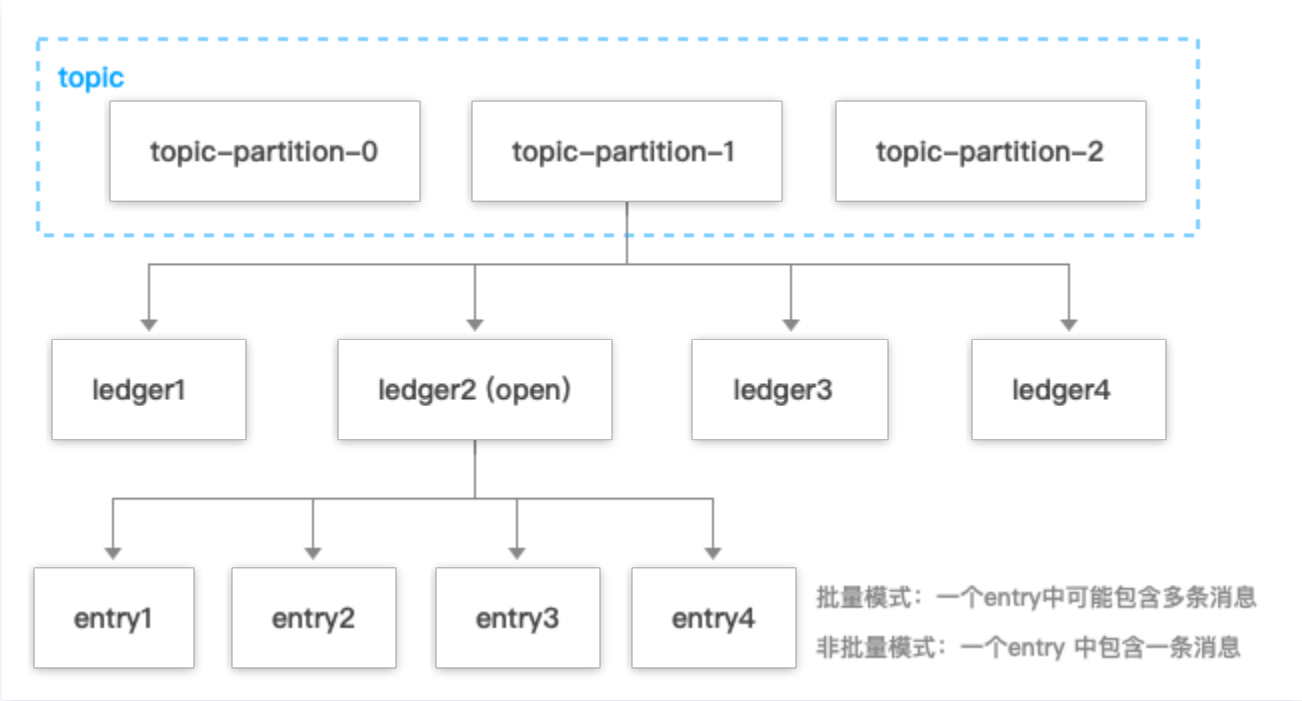
最近更新时间：2022-05-27 10:09:21

消息 ID 生成规则

在 Pulsar 中，每条消息都有自己的 ID（即 MessageID），MessageID 由四部分组成：`ledgerId:entryID:partition-index:batch-index` 。其中：

- partition-index：指分区的编号，在非分区 topic 的时候为 -1。
- batch-index：在非批量消息的时候为 -1。

消息 ID 的生成规则由 Pulsar 的消息存储机制决定，Pulsar 中消息存储原理图如下：



如上图所示，在 Pulsar中，一个 Topic 的每一个分区会对应一系列的 ledger，其中只有一个 ledger 处于 open 状态即可写状态，而每个 ledger 只会存储与之对应的分区下的消息。
Pulsar 在存储消息时，会先找到当前分区使用的 ledger ，然后生成当前消息对应的 entry ID，entry ID 在同一个 ledger 内是递增的。每个 ledger 存在的时长或保存的 entry 个数超过阈值后会进行切换，新的消息会存储到同一个 partition 中的下一个 ledger 中。

- 批量生产消息情况下，一个 entry 中可能包含多条消息。
- 非批量生产的情况下，一个 entry 中包含一条消息（producer 端可以配置这个参数，默认是批量的）。

Ledger 只是一个逻辑概念，是数据的一种逻辑组装维度，并没有对应的实体。而 bookie 只会按照 entry 维度进行写入、查找、获取。

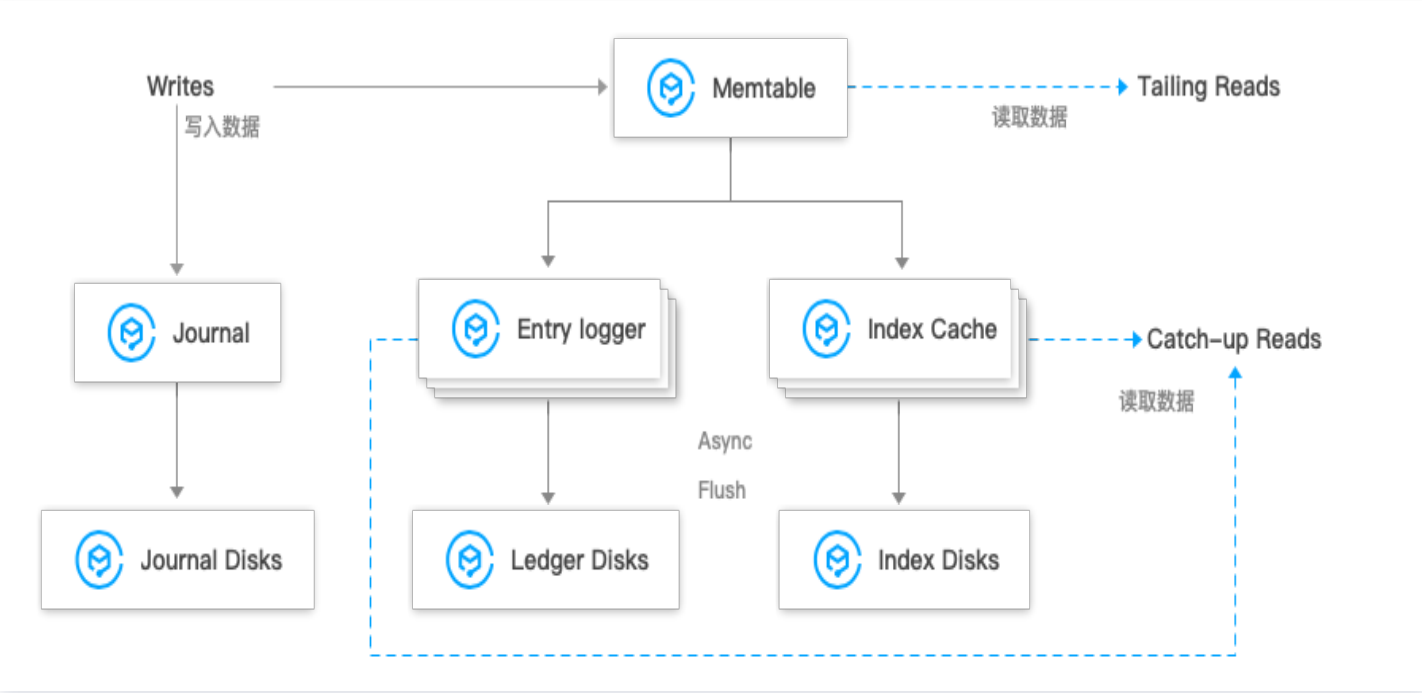
分片机制详解：Ledger 和 Entry

Pulsar 中的消息数据以 ledger 的形式存储在 BookKeeper 集群的 bookie 存储节点上。Ledger 是一个只追加的数据结构，并且只有一个写入器，这个写入器负责多个 bookie 的写入。Ledger 的条目会被复制到多个 bookie 中，同时会写入相关的数据来保证数据的一致性。

BookKeeper 需要保存的数据包括：

- **Journals**
  - journals 文件里存储了 BookKeeper 的事务日志，在任何针对 ledger 的更新发生前，都会先将这个更新的描述信息持久化到这个 journal 文件中。
  - BookKeeper 提供有单独的 sync 线程根据当前 journal 文件的大小来作 journal 文件的 rolling。
- **EntryLogFile**
  - 存储真正数据的文件，来自不同 ledger 的 entry 数据先缓存在内存buffer中，然后批量flush到EntryLogFile中。
  - 默认情况下，所有ledger的数据都是聚合然后顺序写入到同一个EntryLog文件中，避免磁盘随机写。
- **Index 文件**
  - 所有 Ledger 的 entry 数据都写入相同的 EntryLog 文件中，为了加速数据读取，会作 ledgerId + entryId 到文件 offset 的映射，这个映射会缓存在内存中，称为 IndexCache。
  - IndexCache 容量达到上限时，会被 sync 线程 flush 到磁盘中。

三类数据文件的读写交互如下图：



Entry 数据写入

1. 数据首先会同时写入 Journal（写入 Journal 的数据会实时落到磁盘）和 Memtable（读写缓存）。
2. 写入 Memtable 之后，对写入请求进行响应。
3. Memtable 写满之后，会 flush 到 Entry Logger 和 Index cache，Entry Logger 中保存数据，Index cache 中保存数据的索引信息，

4. 后台线程将 Entry Logger 和 Index cache 数据落到磁盘。

Entry 数据读取

- Tailing read 请求：直接从 Memtable 中读取 Entry。
- Catch-up read（滞后消费）请求：先读取 Index信息，然后索引从 Entry Logger 文件读取 Entry。

数据一致性保证：LastLogMark

- 写入的 EntryLog 和 Index 都是先缓存在内存中，再根据一定的条件周期性的 flush 到磁盘，这就造成了从内存到持久化到磁盘的时间间隔，如果在这间隔内 BookKeeper 进程崩溃，在重启后，我们需要根据 journal 文件内容来恢复，这个 LastLogMark 就记录了从 journal 中什么位置开始恢复。
- 它其实是存在内存中，当 IndexCache 被 flush 到磁盘后其值会被更新，LastLogMark 也会周期性持久化到磁盘文件，供 Bookkeeper 进程启动时读取来从 journal 中恢复。
- LastLogMark 一旦被持久化到磁盘，即意味着在其之前的 Index 和 EntryLog 都已经被持久化到了磁盘，那么 journal 在这 LastLogMark 之前的数据都可以被清除了。

消息副本与存储机制

最近更新时间：2023-11-16 10:07:58

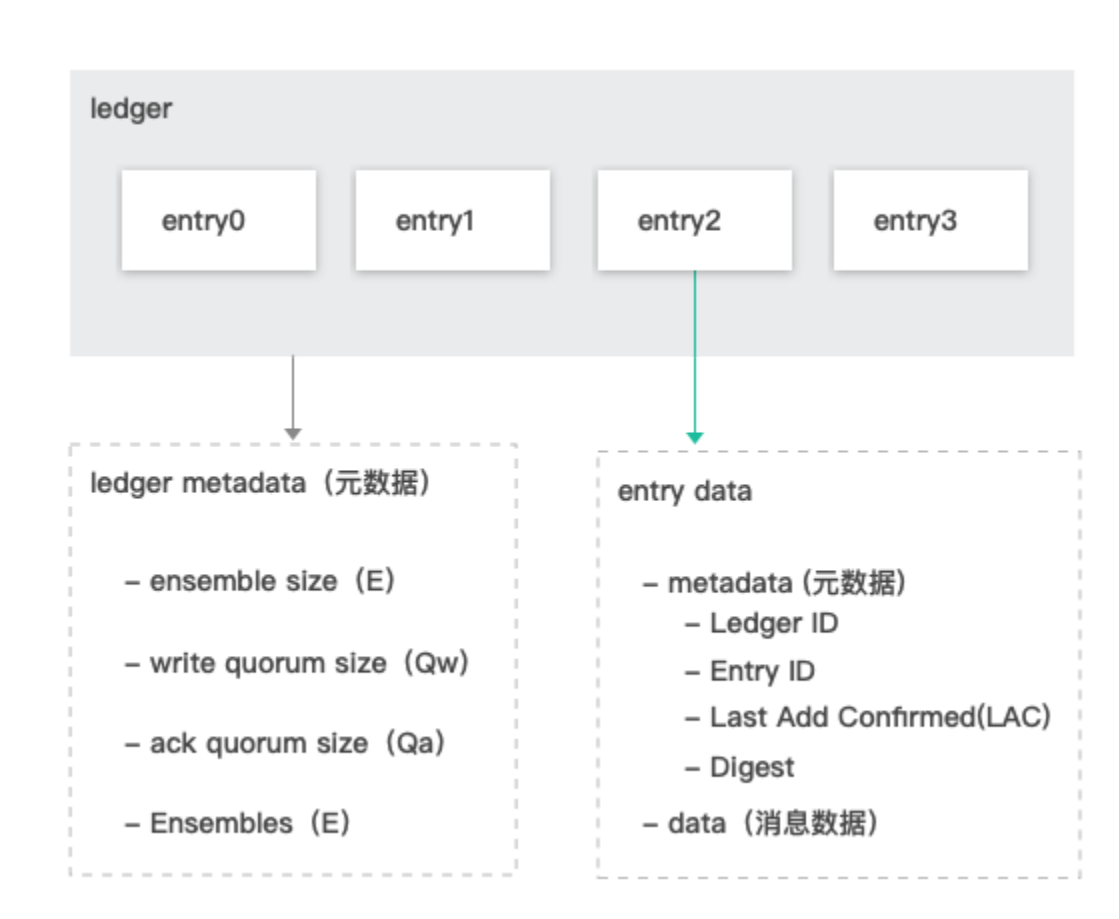
消息元数据组成

Pulsar 中每个分区 Topic 的消息数据以 ledger 的形式存储在 BookKeeper 集群的 bookie 存储节点上，每个 ledger 包含一组 entry，而 bookie 只会按照 entry 维度进行写入、查找、获取。

> **说明**
> 批量生产消息的情况下，一个 entry 中可能包含多条消息，所以 entry 和消息并不一定是一一对应的。

Ledger 和 entry 分别对应不同的元数据。

- ledger 的元数据存储在 zk 上。
- entry 除了消息数据部分之外，还包含元数据，entry 的数据存储在 bookie 存储节点上。



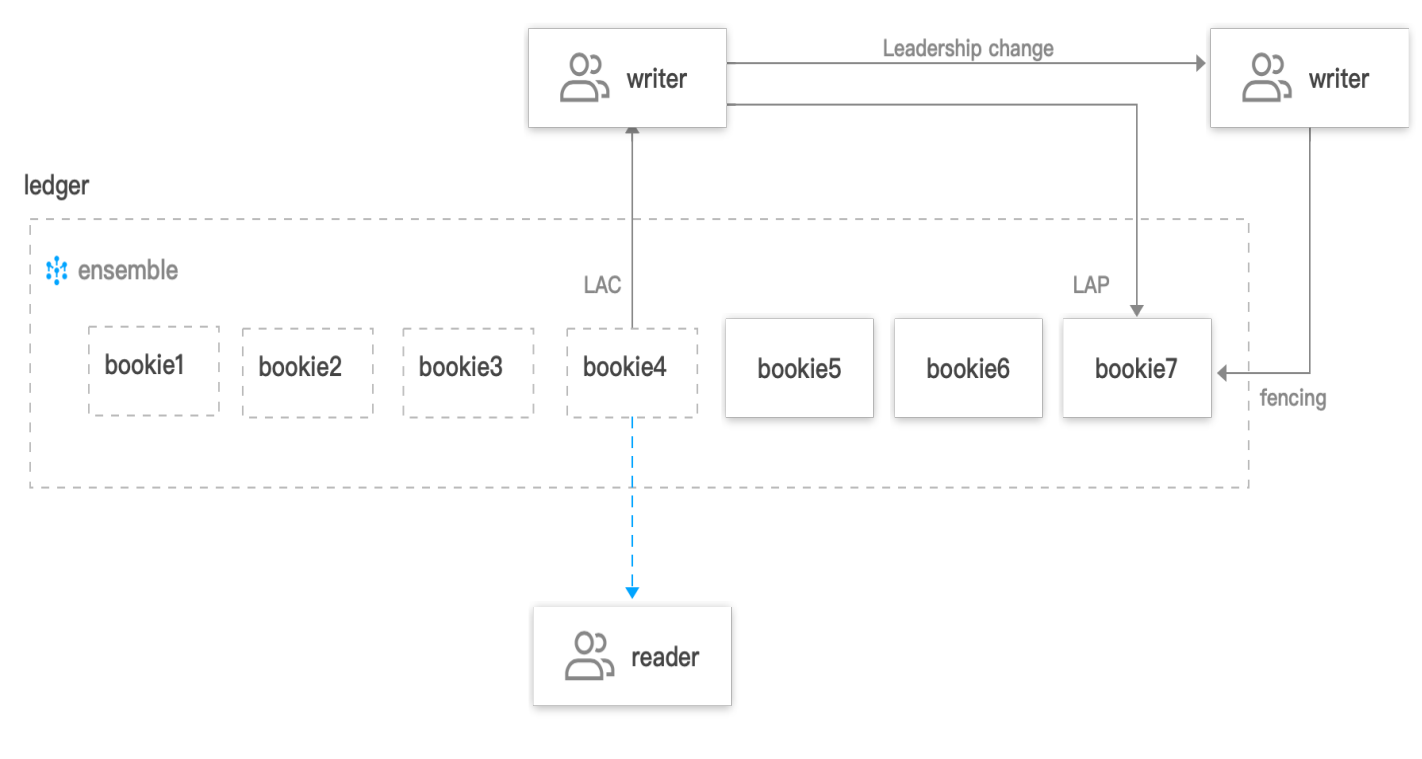
类型	参数	参数说明	数据存放位置
ledger	ensemble size（E）	每个 ledger 选用的 bookie 节点的个数	元数据存储在 zk 上

	write quorum size（Qw）	每个 entry 需要向多少个 bookie 发送写入请求	
	ack quorum size（Qa）	收到多少个写入确认后，即可认为写入成功	
	Ensembles（E）	使用的 ensemble 列表，形式为<entry id,="" ensembles=""> 元组 key（entry id）：使用这个 ensembles 列表开始时的 entry id value（ensembles）：ledger 选用的 bookie ip 列表，每个 value 中包含 ensemble size（E）个 IP 每个 ledger 可能包含多个 ensemble 列表，同一时刻每个 ledger 最多只有一个 ensembles 列表在使用	
Entry	Ledger ID	entry 所在的 ledger id	数据存储在 bookie 存储节点上
	Entry ID	当前 entry id	
	Last Add Confirmed	创建当前 entry 的时候，已知最新的写入确认的 entry id	
	Digest	CRC	

每个 ledger 在创建的时候，会在现有的 BookKeeper 集群中的可写状态的 bookie 候选节点列表中，选用 ensemble size 对应个数的 bookie 节点，如果没有足够的候选节点则会抛出 BKNotEnoughBookiesException 异常。选出候选节点后，将这些信息组成 <entry id, ensembles> 元组，存储到 ledger 的元数据里的 ensembles 中。

消息副本机制

消息写入流程



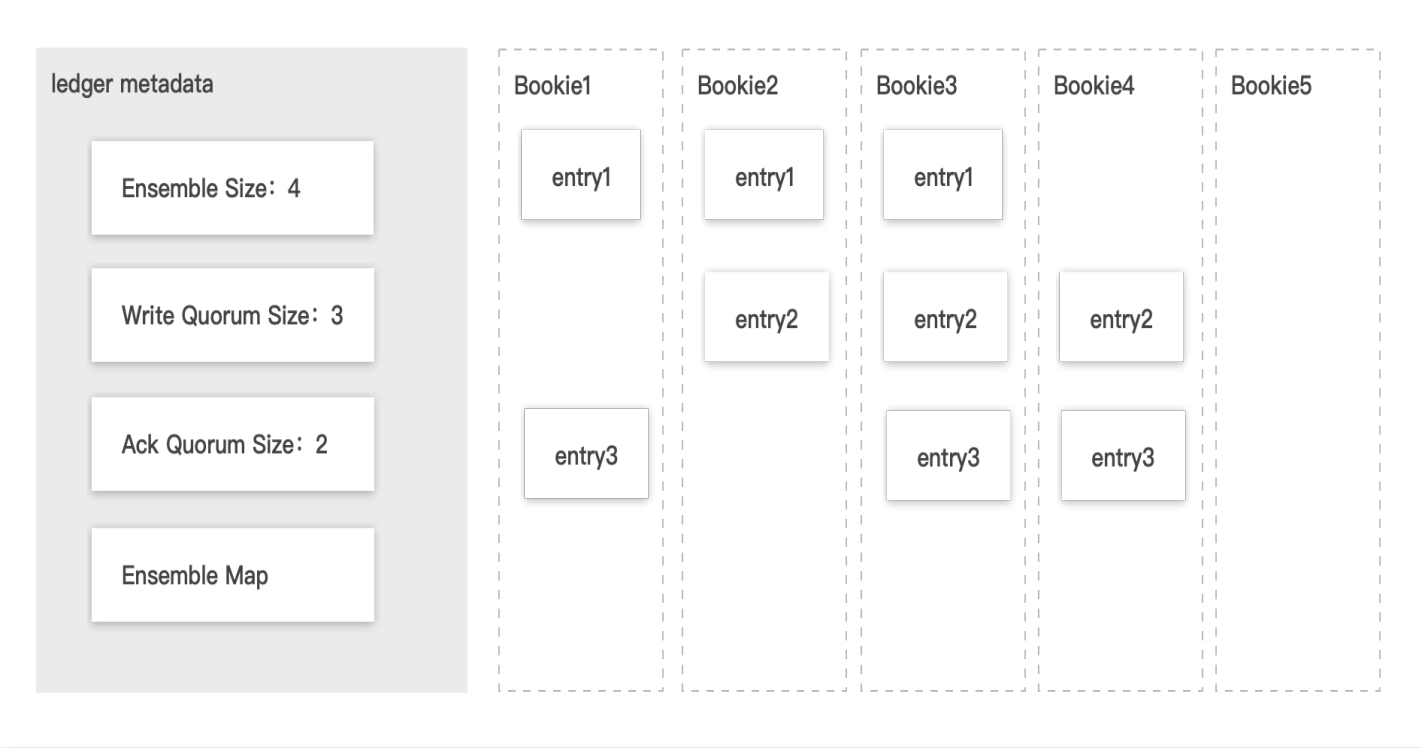
客户端在写入消息时，每个 entry 会向 ledger 当前使用的 ensemble 列表中的 Qw 个 bookie 节点发送写入请求，当收到 Qa 个写确认后，即认为当前消息写入存储成功。同时会通过 LAP（lastAddPushed）和 LAC（LastAddConfirmed）分别标识当前推送的位置和已经收到存储确认的位置。

每个正在推送的 entry 中的 LAC 元数据值，为当前时刻创建发送 entry 请求时，已经收到最新的确认位置值。LAC 所在位置及之前的消息对读客户端是可见的。

同时，pulsar 通过 fencing 机制，来避免同时有多个客户端对同一个 ledger 进行写操作。这里主要适用于一个 topic/partition 的归属关系从一个 broker 变迁到另一个 broker 的场景。

消息副本分布

每个 entry 写入时，会根据当前消息的 entry id 和当前使用的 ensembles 列表的开始 entry id（即key值），计算出在当前 entry 需要使用 ensemble 列表中由哪组 Qw 个 bookie 节点进行写入。之后，broker 会向这些 bookie 节点发送写请求，当收到 Qa 个写确认后，即认为当前消息写入存储成功。这时至少能够保证 Qa 个消息的副本个数。



如上图所示，ledger 选用了4个 bookie 节点（bookie1-4 这4个节点），每次写入3个节点，当收到2个写入确认即代表消息存储成功。当前 ledger 选中的 ensemble 从 entry 1开始，使用 bookie1、bookie2、bookie3 进行写入，写入 entry 2的时候选用 bookie2、bookie3、bookie4 写入，而 entry 3 则会根据计算结果，写入 bookie3、bookie4、bookie1。

消息恢复机制

Pulsar 的 BookKeeper 集群中的每个 bookie 在启动的时候，默认自动开启 recovery 的服务，这个服务会进行如下几个事情：

1. auditorElector 审计选举。
2. replicationWorker 复制任务。
3. deathWatcher 宕机监控。

BookKeeper 集群中的每个 bookie 节点，会通过 zookeeper 的临时节点机制进行选主，主 bookie 主要处理如下几个事情：

1. 负责监控 bookie 节点的变化。
2. 到 zk 上面标记出宕机的 bookie 上面的 ledger 为 Underreplicated 状态。
3. 检查所有的 ledger 的副本数（默认一周一个周期）。
4. Entry 副本数检查（默认未开启）。

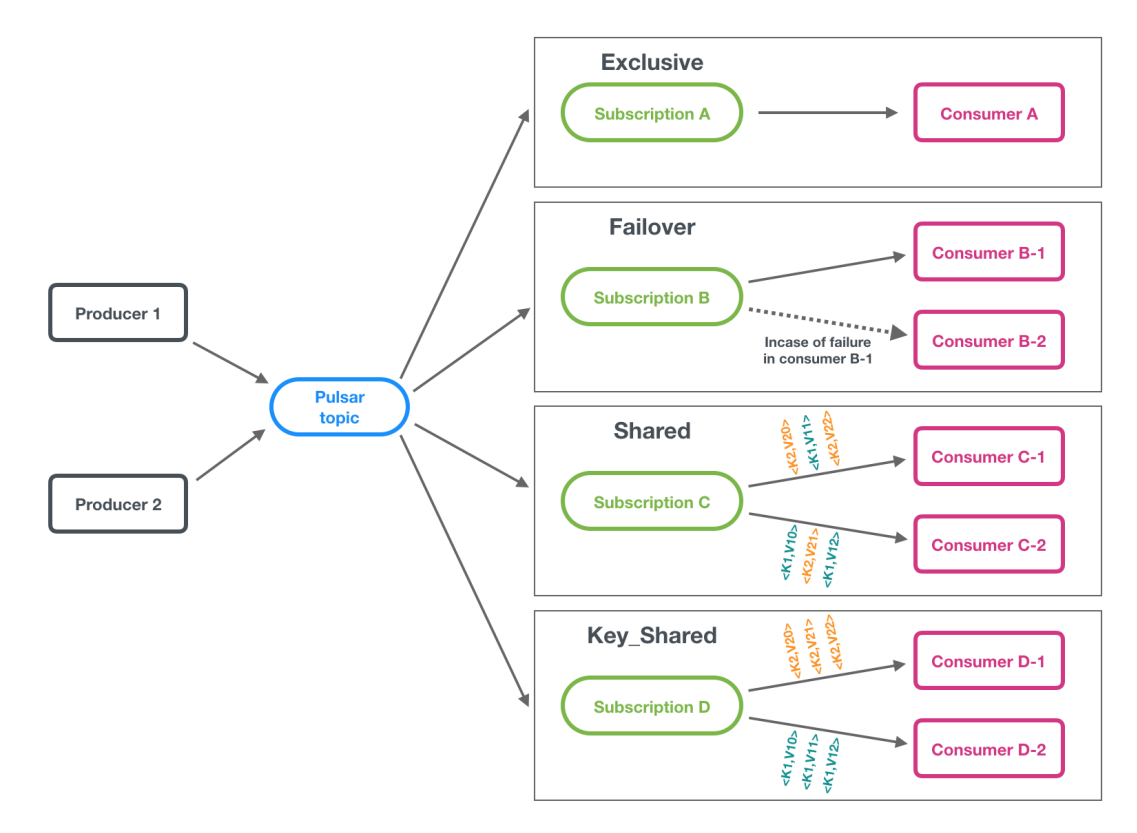
其中 ledger 中的数据是按照 Fragment 维度进行恢复的（每个 Fragment 对应 ledger 下的一组 ensemble 列表，如果一个 ledger 下有多个 ensemble 列表，则需要处理多个 Fragment）。

在进行恢复时，首先要判断出当前的 ledger 中的哪几个 Fragment 中的哪些存储节点需要用新的候选节点进行替换和恢复数据。当 Fragment 中关联的部分 bookie 节点上面没有对应的 entry 数据（默认是按照首、尾 entry 是否存在判断），则这个 bookie 节点需要被替换，当前的这个 Fragment 需要进行数据恢复。
Fragment 的数据用新的 bookie 节点进行数据恢复完毕后，更新 ledger 的元数据中当前 Fragment 对应的 ensemble 列表的原数据。
经过此过程，因 bookie 节点宕机引起的数据副本数减少的场景，数据的副本数会逐步的恢复成 Qw（后台指定的副本数，TDMQ 默认3副本）个。

使用实践
订阅模式

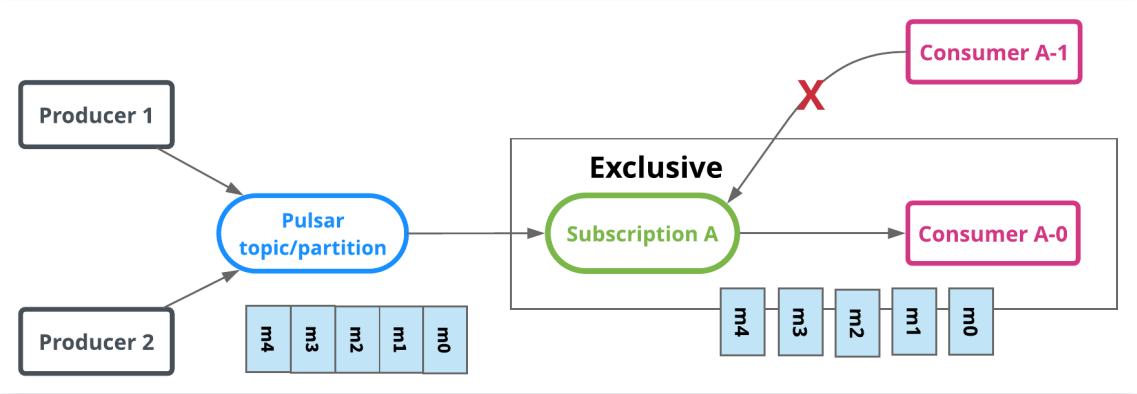
最近更新时间：2023-05-09 16:27:53

为了适用不同场景的需求，Pulsar 支持四种订阅模式：Exclusive、Shared、Failover、Key_Shared。



独占模式（Exclusive）

Exclusive 独占模式（默认模式）：一个 Subscription 只能与一个 Consumer 关联，只有这个 Consumer 可以接收到 Topic 的全部消息，如果该 Consumer 出现故障了就会停止消费。
Exclusive 订阅模式下，同一个 Subscription 里只有一个 Consumer 能消费 Topic，如果多个 Consumer 订阅则会报错，适用于全局有序消费的场景。



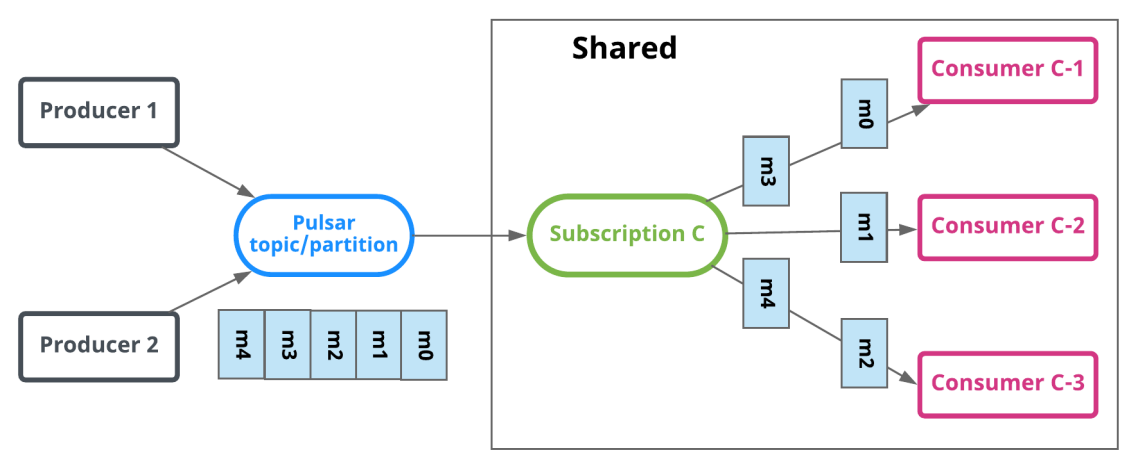

```
// 构建消费者
Consumer<byte[]> consumer = pulsarClient.newConsumer()
    // topic完整路径，格式为persistent://集群（租户）ID/命名空间/Topic名称，从【Topic管理】处复制
    .topic("persistent://pulsar-xxx/sdk_java/topic1")
    // 需要在控制台Topic详情页创建好一个订阅，此处填写订阅名
    .subscriptionName("sub_topic1")
    // 声明消费模式为exclusive（独占）模式
    .subscriptionType(SubscriptionType.Exclusive)
    .subscribe();
```

启动多个消费者将收到错误信息如下图所示：

```
\java.exe ...
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.                                    for further details.
>> pulsar client created.
Exception in thread "main" org.apache.pulsar.client.api.PulsarClientException$ConsumerBusyException: Failed to subscribe persistent
Exclusive consumer is already connected
    at org.apache.pulsar.client.api.PulsarClientException.unwrap(PulsarClientException.java:946)
    at org.apache.pulsar.client.impl.ConsumerBuilderImpl.subscribe(ConsumerBuilderImpl.java:102)
    at com.tencent.cloud.tdmq.pulsar.simple.ListenerConsumer.main(ListenerConsumer.java:41)
```

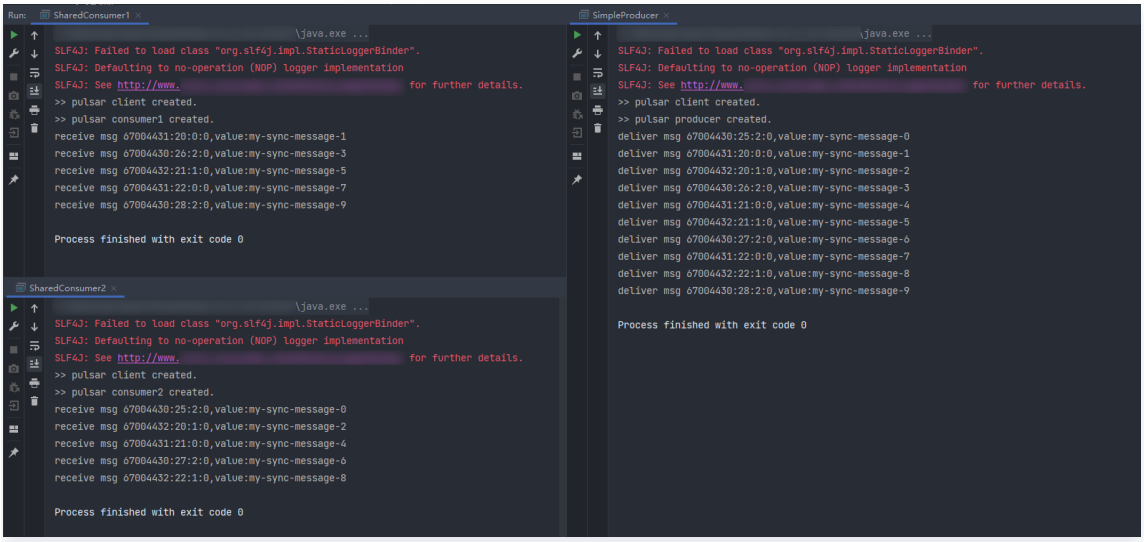
共享模式（Shared）

消息通过 round robin 轮询机制（也可以自定义）分发给不同的消费者，并且每个消息仅会被分发给一个消费者。当消费者断开连接，所有被发送给他，但没有被确认的消息将被重新安排，分发给其它存活的消费者。



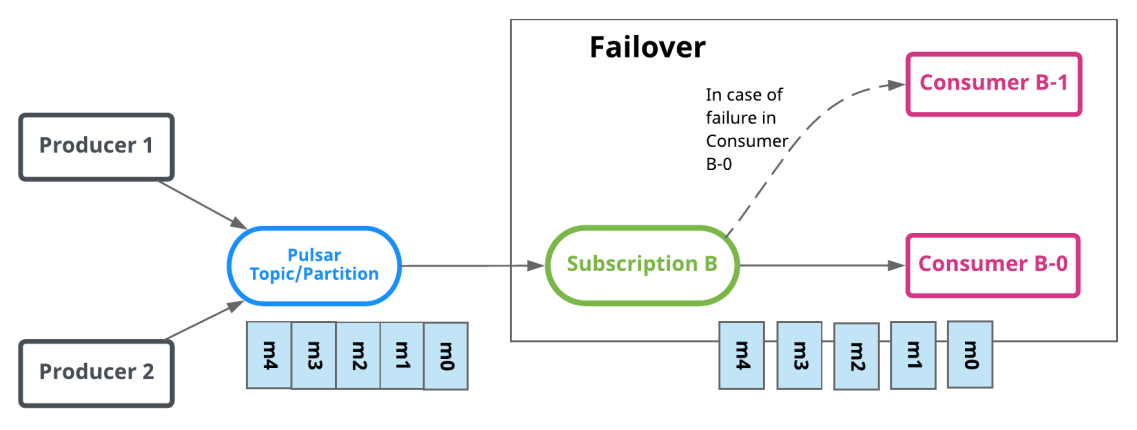

```
// 构建消费者
Consumer<byte[]> consumer = pulsarClient.newConsumer()
    // topic完整路径，格式为persistent://集群（租户）ID/命名空间/Topic名称，从【Topic管理】处复制
    .topic("persistent://pulsar-xxx/sdk_java/topic1")
    // 需要在控制台Topic详情页创建好一个订阅，此处填写订阅名
    .subscriptionName("sub_topic1")
    // 声明消费模式为 Shared（共享）模式
    .subscriptionType(SubscriptionType.Shared)
    .subscribe();
```

多个 Shared 模式消费者如下图所示：



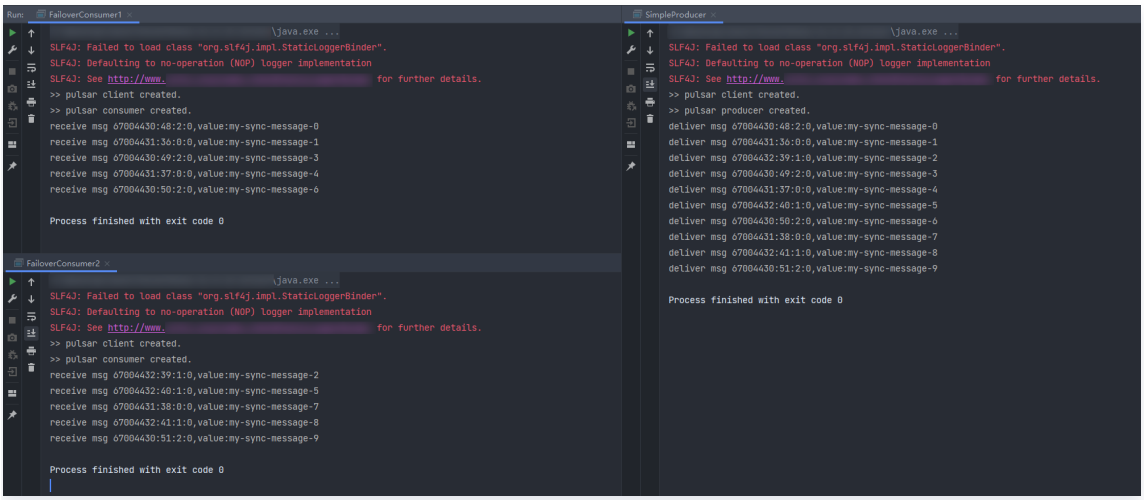
灾备模式（Failover）

当存在多个 consumer 时，将会按字典顺序排序，第一个 consumer 被初始化为唯一接受消息的消费者。当第一个 consumer 断开时，所有的消息（未被确认和后续进入的）将会被分发给队列中的下一个 consumer。



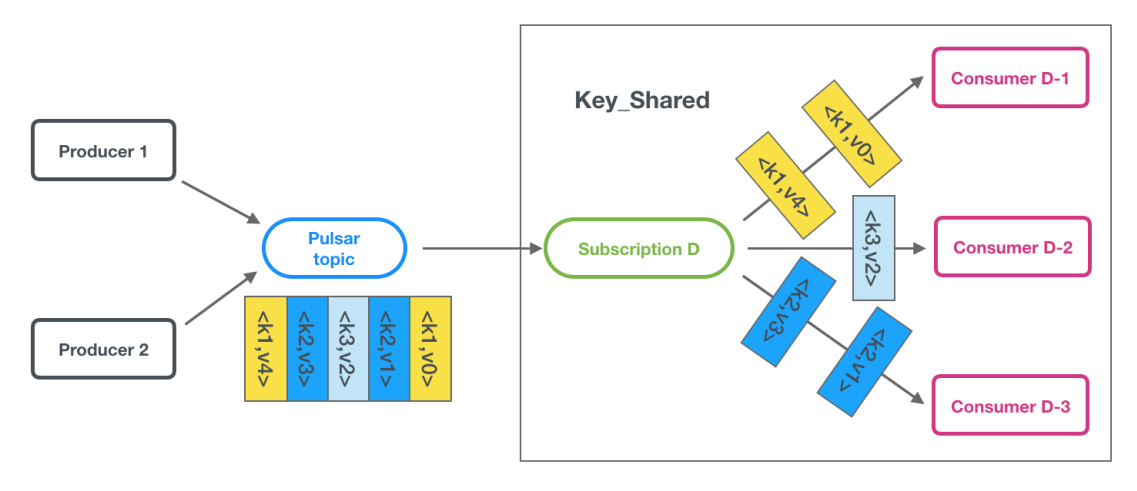

```
// 构建消费者
Consumer<byte[]> consumer = pulsarClient.newConsumer()
    // topic完整路径，格式为persistent://集群（租户）ID/命名空间/Topic名称，从【Topic管理】处复制
    .topic("persistent://pulsar-xxx/sdk_java/topic1")
    // 需要在控制台Topic详情页创建好一个订阅，此处填写订阅名
    .subscriptionName("sub_topic1")
    // 声明消费模式为灾备模式
    .subscriptionType(SubscriptionType.Failover)
    .subscribe();
```

多个 Failover 模式消费者如下图所示：



KEY 共享模式（Key_Shared）

当存在多个 Consumer 时，将根据消息的 Key 进行分发，Key 相同的消息只会被分发到同一个消费者。



注意：

- Key_Shared 本身在使用上存在一定的限制条件，由于其工程实现复杂度较高，在社区版本迭代中，不断有对 Key_Shared 的功能进行改进以及优化，整体稳定性相较 Exclusive,Failover 和 Shared 这三种订阅类型偏弱。如果上述三种订阅类型能满足业务需要，可以优先选用上述三种订阅类型。
- 专业集群可以保证相同 KEY 的消息按顺序投递；虚拟集群无法保障消息投递顺序。

Key_Shared 使用建议

什么时候才考虑用 Key_Shared 订阅模式

如是普通的生产消费场景，建议直接选用 Shared 模式即可。
若需要让相同 Key 的消息分给同一个消费者，这个时候 Shared 订阅模式无法满足用户需求。有两种方式可以选择：

- 选择 Key_Shared 订阅模式。
- 通过多分区主题 + Failover 订阅模式实现。

什么场景下适合用 Key_Shared 订阅

- Key 数量多且每个 Key 的消息分布相对均匀
- 消费处理速度快，无消息堆积的情况

如果在生产过程中不能保证上面的两个条件同时满足，建议用【多分区主题 + Failover 订阅】

代码示例

Key_Shared 订阅示例

默认情况下，Pulsar 在生产消息时是开启 Batch 功能的，Pulsar 的 Batch 消息解析是在 Consumer 侧处理的。所以在 Broker 侧一个 Batch 消息是被当作一条 Entry 处理的，所以对于 Key_shared 的基于消息 Key

有序订阅类型来说，是没办法处理这种 Case 的，因为不同 Key 的消息有可能被打包到同一个 Batch 中。针对这种情况在创建 Producer 时有如下两种规避方式：

1. 禁用 Batch。

```
// 构建生产者
Producer<byte[]> producer pulsarClient.newProducer()
        .topic(topic)
        .enableBatching(false)
        .create();
// 发送消息时设置key
MessageId msgId = producer.newMessage()
 // 消息内容
 .value(value.getBytes(StandardCharsets.UTF_8))
 // 在此处设置key，key相同的消息只会被分发到同一个消费者。
 .key("youKey1")
 .send();
```

2. 使用 key_based batch 类型。

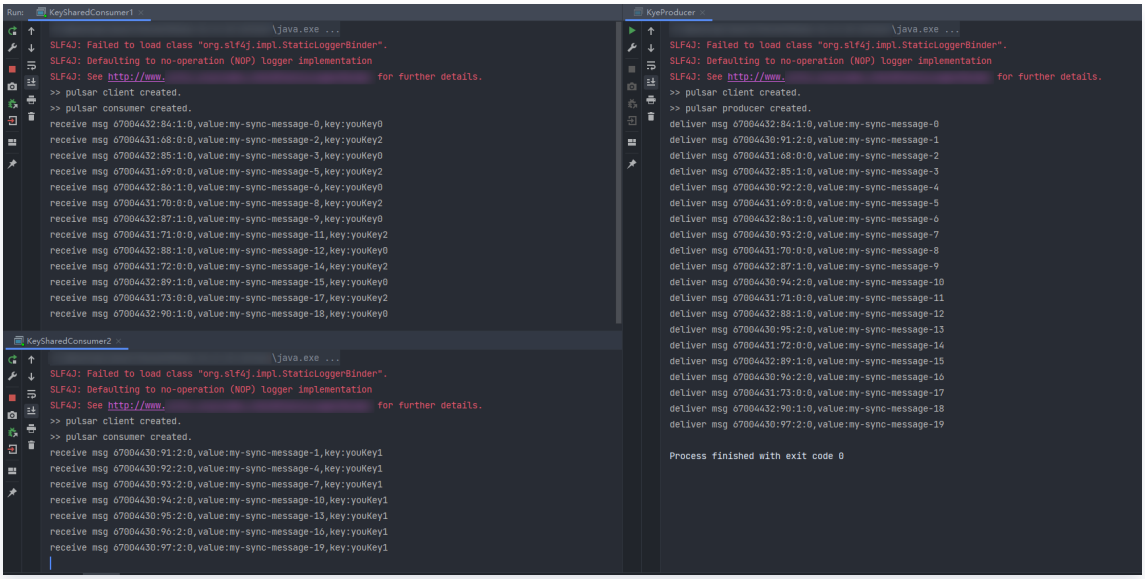
```
// 构建生产者
Producer<byte[]> producer = pulsarClient.newProducer()
        .topic(topic)
        .enableBatching(true)
        .batcherBuilder(BatcherBuilder.KEY_BASED)
        .create();
// 发送消息时设置key
MessageId msgId = producer.newMessage()
 // 消息内容
 .value(value.getBytes(StandardCharsets.UTF_8))
 // 在此处设置key，key相同的消息只会被分发到同一个消费者。
 .key("youKey1")
 .send();
```

消费者代码示例：

```
// 构建消费者
Consumer<byte[]> consumer = pulsarClient.newConsumer()
   // topic完整路径，格式为persistent://集群（租户）ID/命名空间/Topic名称，从【Topic管
理】处复制
   .topic("persistent://pulsar-xxx/sdk_java/topic1")
   // 需要在控制台Topic详情页创建好一个订阅，此处填写订阅名
   .subscriptionName("sub_topic1")
   // 声明消费模式为 Key_Shared（Key 共享）模式
```

```
    .subscriptionType(SubscriptionType.Key_Shared)
    .subscribe();
```

多个 Key_Shared 模式消费者。



多分区主题 + Failover 订阅示例

注意事项：

- 在该模式下，每个分区同时只会分配给一个消费者实例。若消费者数量多于分区数量，超出数量的消费者无法参与消息，可以通过扩容分区数量不小于消费者数量解决。
- 在设计 Key 的时候尽量保证 Key 分布均匀。
- Failover 模式下不支持延时消息。

1. 生产者代码示例

```
// 构建生产者
Producer<byte[]> producer pulsarClient.newProducer()
            .topic(topic)
            .enableBatching(false)  // 禁用batch
            .create();
// 发送消息时设置key
MessageId msgId = producer.newMessage()
 // 消息内容
 .value(value.getBytes(StandardCharsets.UTF_8))
 // 在此处设置key，key相同的消息会发送到同一个分区中
 .key("youKey1")
 .send();
```

2. 消费者代码示例

```
// 构建消费者
Consumer<byte[]> consumer = pulsarClient.newConsumer()
    // topic完整路径，格式为persistent://集群（租户）ID/命名空间/Topic名称，从【Topic管理】处复制
    .topic("persistent://pulsar-xxx/sdk_java/topic1")
    // 需要在控制台Topic详情页创建好一个订阅，此处填写订阅名
    .subscriptionName("sub_topic1")
    // 声明消费模式为Failover模式
    .subscriptionType(SubscriptionType.Failover)
    .subscribe();
```

开启保序

TDMQ Pulsar 2.9.2 版本集群可以支持 KEY 的顺序投递。如需开启，需要在创建消费者实例时指定 keySharedPolicy。

```
// 构建消费者
Consumer<byte[]> consumer = pulsarClient.newConsumer()
    // topic完整路径，格式为persistent://集群（租户）ID/命名空间/Topic名称，从【Topic管理】处复制
    .topic("persistent://pulsar-xxx/sdk_java/topic1")
    // 需要在控制台Topic详情页创建好一个订阅，此处填写订阅名
    .subscriptionName("sub_topic1")
    // 声明消费模式为 Key_Shared（Key 共享）模式
    .subscriptionType(SubscriptionType.Key_Shared)
    // 设置为不允许乱序
    .keySharedPolicy(KeySharedPolicy.autoSplitHashRange().setAllowOutOfOrderDelivery(false))
    .subscribe();
```

注意：

- 集群 2.7.2 版本不支持开启保序，可能造成消息推送堵塞，无法消费；
- 开启了保序，消费者重启后可能会出现消费速率下降、消息堆积的情况，这是因为在保序模式下新消费者上线后，需要等待消费者上线之前的全部消息都被消费完成后（全部确认后），才能继续消费后面的消息。

定时和延时消息

最近更新时间：2024-02-01 10:50:11

相关概念

- **定时消息**：消息在发送至服务端后，实际业务并不希望消费端马上收到这条消息，而是推迟到某个时间点被消费，这类消息统称为定时消息。
- **延时消息**：消息在发送至服务端后，实际业务并不希望消费端马上收到这条消息，而是推迟一段时间后再被消费，这类消息统称为延时消息。

实际上，定时消息可以看成是延时消息的一种特殊用法，其实现的最终效果和延时消息是一致的。

适用场景

如果系统是一个单体架构，则通过业务代码自己实现延时或利用第三方组件实现基本没有差别；一旦架构复杂起来，形成了一个大型分布式系统，有几十上百个微服务，这时通过应用自己实现定时逻辑会带来各种问题。一旦运行着延时程序的某个节点出现问题，整个延时的逻辑都会受到影响。
针对以上问题，利用延时消息的特性投递到消息队列里，便是一个较好的解决方案，能统一计算延时时间，同时重试和死信机制确保消息不丢失。
具体场景的示例如下：

- 微信红包发出后，生产端发送一条延时24小时的消息，到了24小时消费端程序收到消息，进行用户是否已经领走红包的判断，如果没有则退还到原账户。
- 小程序下单某商品后，后台存放一条延时30分钟的消息，到时间之后消费端收到消息触发对支付结果的判断，如果没有支付就取消订单，这样就实现了超过30分钟未完成支付就取消订单的逻辑。
- 微信上用户将某条信息设置待办后，也可以通过发送一条定时消息，服务端到点主动消费这条定时消息，对用户进行待办项提醒。

使用方式

在 TDMQ Pulsar 版的 SDK 中提供了专门的 API 来实现定时消息和延时消息。

- 对于定时消息，您需要提供一个消息发送的时刻。
- 对于延时消息，您需要提供一个时间长度作为延时的时长。

定时消息

定时消息通过生产者 `producer` 的 deliverAt() 方法实现，代码示例如下：

```
String value = "message content";
try {
        //需要先将显式的时间转换为 Timestamp
    long timeStamp = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").parse("2020-11-11 00:00:00").getTime();
```

```
    //通过调用 producer 的 deliverAt 方法来实现定时消息
    MessageId msgId = producer.newMessage()
            .value(value.getBytes())
            .deliverAt(timeStamp)
            .send();
} catch (ParseException e) {
    //TODO 添加对 Timestamp 解析失败的处理方法
    e.printStackTrace();
}
```

注意

- 定时消息的时间范围为当前时间开始计算，864000秒（10天）以内的任意时刻。如10月1日12:00开始，最长可以设置到10月11日12:00。
- 定时消息不可以使用 batch 模式发送，请在创建 producer 的时候把 `enableBatch` 参数设为 `false` 。
- 定时消息的消费模式仅支持使用 Shared 模式进行消费，否则会失去定时效果（Key-shared 也不支持）。

延时消息

延时消息通过生产者 `produce` 的 deliverAfter() 方法实现，代码示例如下：

```
String value = "message content";

//需要指定延时的时长
long delayTime = 10L;
//通过调用 producer 的 deliverAfter 方法来实现定时消息
MessageId msgId = producer.newMessage()
    .value(value.getBytes())
    .deliverAfter(delayTime, TimeUnit.SECONDS) //单位可以自由选择
    .send();
```

注意

- 延时消息的时长取值范围为0 – 864000秒（0秒 – 10天）。如10月1日12:00开始，最长可以设置864000秒。
- Go SDK 当延时消息时间超过 10 天时，将可能出现重复消息的情况。
- 延时消息不可以使用 batch 模式发送，请在创建 producer 的时候把 `enableBatch` 参数设为 `false` 。
- 延时消息的消费模式仅支持使用 Shared 模式进行消费，否则会失去延时效果（Key-shared 也不支

持）。

使用说明和限制

- 使用定时或延迟消息时，建议与普通消息使用不同的 Topic 来管理，即定时与延迟消息发送到一个固定的 Topic，普通消息发送到另一个 Topic 中，方便后续的管理与维护，增加稳定性。
- 使用定时和延时两种类型的消息时，请确保客户端的机器时钟和服务端的机器时钟（所有地域均为UTC+8 北京时间）保持一致，否则会有时差。
- 定时和延时消息在精度上会有1秒左右的偏差。
- 定时和延时消息不支持 batch 模式（批量发送），batch 模式会引起消息堆积，保险起见，请在创建 producer 的时候把 `enableBatch` 参数设为 `false` 。
- 定时和延时消息的消费模式仅支持使用 Shared 模式进行消费，否则会失去定时或延时效果（Key-shared 也不支持）。
- 关于定时和延时消息的时间范围，最大均为10天。
- 使用定时消息时，设置的时刻在当前时刻以后才会有定时效果，否则消息将被立即发送给消费者。
- 设定定时时间后，TTL 的时间依旧会从发送消息的时间点开始算消息的最长保留时间；例如定时到2天后发送，消息最长保留（TTL）如果设置为1天的话，则消息在1天后会被删除，这个时候要确保 TTL 的时间要大于延时的时间，即 TTL 设置成大于等于2天，否则 TTL 到期时，消息会被删除。延时消息同理。
- 普通类型 Topic 支持收发定时/延时消息，调用 使用方式 中的 API 即可发送定时/延时消息。

消息标签过滤

最近更新时间：2024-02-21 15:41:11

本文主要介绍 TDMQ Pulsar 版中消息标签过滤的功能、应用场景和使用方式。

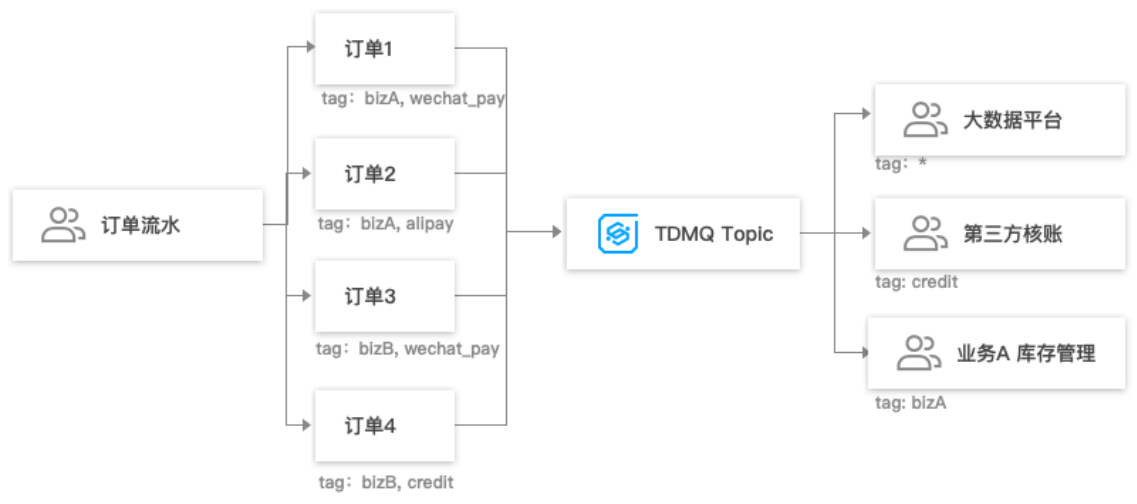
功能介绍

Tag，即消息标签，用于对某个Topic下的消息进行分类。TDMQ Pulsar 版的生产者在发送消息时，指定消息的 Tag，消费者需根据已经指定的 Tag 来进行订阅。
消费者订阅 Topic 时若未设置 Tag，Topic 中的所有消息都将被投递到消费端进行消费。

> **注意**
> 在一个订阅中，单个消费者可以使用多个 Tag，多个 Tag 之间的关系是「或」，不同消费者使用的 Tag 需要是相同的。

应用场景

通常，一个 Topic 中存放的是相同业务属性的消息，例如交易流水 Topic 包含了下单流水、支付流水、发货流水等，业务若只想消费者其中一种类别的流水，可在客户端进行过滤，但这种过滤方式会带来带宽的资源浪费。
针对上述场景，TDMQ Pulsar 提供 Broker 端过滤的方式，用户可在生产消息时设置一个或者多个 Tag 标签，消费时指定 Tag 订阅。



使用说明

Tag 消息目前是通过 Properties 的方式传入的，可以通过如下方式获取：

Java

```
<dependency>
    <groupId>org.apache.pulsar</groupId>
    <artifactId>pulsar-client</artifactId>
    <version>2.10.1</version> <!-- 推荐版本 -->
</dependency>
```

Go

推荐使用 0.9.0 及以上版本。

```
go get -u github.com/apache/pulsar-client-go@master
```

Tag 消息使用限制

- Tag 消息不支持 Batch 功能，Batch 功能默认是开启的。如果要使用 Tag 消息，需要在 Producer 侧禁用掉 batch，具体如下：

Java

```
// 构建生产者
Producer<byte[]> producer = pulsarClient.newProducer()
        // 禁用掉batch功能
        .enableBatching(false)
        // topic完整路径，格式为persistent://集群（租户）ID/命名空间/Topic名称
        .topic("persistent://pulsar-xxx/sdk_java/topic2").create();
```

Go

```
producer, err := client.CreateProducer(pulsar.ProducerOptions{
    DisableBatching: true, // 禁用掉batch功能
})
```

- tag 消息的过滤只针对已设置 tag 的消息，未设置 tag 的消息，不在过滤范围内。即未设置 tag 的消息会推送给所有的订阅者。
- 如果要开启 Tag 消息，需要发送消息的时候，在 ProducerMessage 中设置 Properties 字段；同时在创建 Consumer 的时候需要在 ConsumerOptions 中指定 SubscriptionProperties 字段。

- 在 ProducerMessage 中设置 Properties 字段时，其中 key 为 tag 的名字，value 为固定值： `TAGS` 。
- 在 ConsumerOptions 中指定 SubscriptionProperties 字段时，其中 key 为要订阅的 tag 的名字，value 为 tag 的版本信息，为保留字段，目前没有实质含义，用来做后续功能的扩展，具体如下：
  - 指定单个 tag

Java

```java
// 发送消息
MessageId msgId = producer.newMessage()
    .property("tag1", "TAGS")
    .value(value.getBytes(StandardCharsets.UTF_8))
      .send();

// 订阅相关参数，可用来设置订阅标签(TAG)
HashMap<String, String> subProperties = new HashMap<>();
 subProperties.put("tag1","1");
 // 构建消费者
 Consumer<byte[]> consumer = pulsarClient.newConsumer()
     // topic完整路径，格式为persistent://集群（租户）ID/命名空间/Topic名称，从【Topic管理】处复制
    .topic("persistent://pulsar-xxxx/sdk_java/topic2")
    // 需要在控制台Topic详情页创建好一个订阅，此处填写订阅名
    .subscriptionName("topic_sub1")
    // 声明消费模式为共享模式
    .subscriptionType(SubscriptionType.Shared)
    // 订阅相关参数，tag订阅等。。
    .subscriptionProperties(subProperties)
     // 配置从最早开始消费，否则可能会消费不到历史消息
     .subscriptionInitialPosition(SubscriptionInitialPosition.Earliest).subscribe();
```

Go

```go
// 发送消息
if msgId, err := producer.Send(ctx, &pulsar.ProducerMessage{
        Payload: []byte(fmt.Sprintf("hello-%d", i)),
  Properties: map[string]string{
      "tag1": "TAGS",
  },
 }); err != nil {
        log.Fatal(err)
  }
```

```
// 创建 consumer
    consumer, err := client.Subscribe(pulsar.ConsumerOptions{
            Topic:           "topic-1",
            SubscriptionName: "my-sub",
            SubscriptionProperties: map[string]string{"tag1": "1"},
     })
```

- 指定多个 tag

Java

```
 // 发送消息
   MessageId msgId = producer.newMessage()
       .property("tag1", "TAGS")
       .property("tag2", "TAGS")
     .value(value.getBytes(StandardCharsets.UTF_8))
       .send();

 // 订阅相关参数，可用来设置订阅标签(TAG)
 HashMap<String, String> subProperties = new HashMap<>();
 subProperties.put("tag1","1");
 subProperties.put("tag2","1");
 // 构建消费者
 Consumer<byte[]> consumer = pulsarClient.newConsumer()
     // topic完整路径，格式为persistent://集群（租户）ID/命名空间/Topic名称，从【Topic管
理】处复制
     .topic("persistent://pulsar-xxxx/sdk_java/topic2")
     // 需要在控制台Topic详情页创建好一个订阅，此处填写订阅名
     .subscriptionName("topic_sub1")
     // 声明消费模式为共享模式
     .subscriptionType(SubscriptionType.Shared)
     // 订阅相关参数，tag订阅等。。
     .subscriptionProperties(subProperties)
     // 配置从最早开始消费，否则可能会消费不到历史消息
     .subscriptionInitialPosition(SubscriptionInitialPosition.Earliest).subscribe();
```

Go

```
 // 创建 producer
 if msgId, err := producer.Send(ctx, &pulsar.ProducerMessage{
         Payload: []byte(fmt.Sprintf("hello-%d", i)),
           Properties: map[string]string{
                 "tag1": "TAGS",
```

```
                "tag2": "TAGS",
            },
    }); err != nil {
            log.Fatal(err)
    }

// 创建 consumer
    consumer, err := client.Subscribe(pulsar.ConsumerOptions{
            Topic:            "topic-1",
            SubscriptionName: "my-sub",
            SubscriptionProperties: map[string]string{
                    "tag1": "1",
                    "tag2": "1",
            },
    })
```

- tag 与 properties 混合

Java

```
  // 发送消息
  MessageId msgId = producer.newMessage()
      .property("tag1", "TAGS")
      .property("tag2", "TAGS")
      .property("xxx", "yyy")
    .value(value.getBytes(StandardCharsets.UTF_8))
      .send();

 // 订阅相关参数，可用来设置订阅标签(TAG)
 HashMap<String, String> subProperties = new HashMap<>();
 subProperties.put("tag1","1");
 subProperties.put("tag2","1");
 // 构建消费者
 Consumer<byte[]> consumer = pulsarClient.newConsumer()
    // topic完整路径，格式为persistent://集群（租户）ID/命名空间/Topic名称，从【Topic管
理】处复制
    .topic("persistent://pulsar-xxxx/sdk_java/topic2")
    // 需要在控制台Topic详情页创建好一个订阅，此处填写订阅名
    .subscriptionName("topic_sub1")
    // 声明消费模式为共享模式
    .subscriptionType(SubscriptionType.Shared)
    // 订阅相关参数，tag订阅等。。
    .subscriptionProperties(subProperties)
    // 配置从最早开始消费，否则可能会消费不到历史消息
```

```
    .subscriptionInitialPosition(SubscriptionInitialPosition.Earliest).subscribe();
```

Go

```
// 创建 producer
if msgId, err := producer.Send(ctx, &pulsar.ProducerMessage{
        Payload: []byte(fmt.Sprintf("hello-%d", i)),
        Properties: map[string]string{
                "tag1": "TAGS",
                "tag2": "TAGS",
                "xxx": "yyy",
        },
    }); err != nil {
        log.Fatal(err)
    }

// 创建 consumer
    consumer, err := client.Subscribe(pulsar.ConsumerOptions{
            Topic:            "topic-1",
            SubscriptionName: "my-sub",
            SubscriptionProperties: map[string]string{
                    "tag1": "1",
                    "tag2": "1",
            },
    })
```

注意

在 consumer 侧设置 SubscriptionProperties 字段时，一旦设定，这个订阅所处理的 tag 信息是不可变更的。如果需要更换订阅的 tag，可以将当前的订阅先 `Unsubscribe` 掉，然后再重新创建新的订阅来处理。

消息重试与死信机制

最近更新时间：2023-08-01 15:25:11

重试 Topic 是一种为了确保消息被正常消费而设计的 Topic 。当某些消息第一次被消费者消费后，没有得到正常的回应，则会进入重试 Topic 中，当重试达到一定次数后，停止重试，投递到死信 Topic 中。
当消息进入到死信队列中，表示 TDMQ Pulsar 版已经无法自动处理这批消息，一般这时就需要人为介入来处理这批消息。您可以通过编写专门的客户端来订阅死信 Topic，处理这批之前处理失败的消息。

自动重试

相关概念

重试 Topic：一个重试 Topic 对应一个订阅名（一个订阅者组的唯一标识），以 Topic 形式存在于 TDMQ Pulsar 版中。当您在控制台新建订阅，并打开**自动创建重试&死信队列**，系统会自动创建重试 Topic，该 Topic 会自主实现消息重试的机制。
该 Topic 命名为：

- 2.9.2 版本集群：[Topic 名称]-[订阅名]-RETRY
- 2.7.2 版本集群：[订阅名]-RETRY
- 2.6.1 版本集群：[订阅名]-retry

实现原理

您创建的消费者使用某个订阅名以共享模式订阅了一个 Topic 后，如果开启了 `enableRetry` 属性，就会自动订阅这个订阅名对应的重试队列。

> **说明**
> - 仅共享模式（包括 Key 共享）支持自动化重试和死信机制，独占和灾备模式不支持。
> - 注意客户端版本需要与集群版本保持一致，客户端才能准确识别自动创建出的重试、死信队列。
> - 当使用 Token 访问重试/死信队列时，需要为消费者所使用角色赋予生产消息权限。

这里以 Java 语言客户端为例，在 `topic1` 创建了一个 `sub1` 的订阅，客户端使用 `sub1` 订阅名订阅了 `topic1` 并开启了 `enableRetry` ，如下所示：

```
Consumer consumer = client.newConsumer()
    .topic("persistent://1******30/my-ns/topic1")
    .subscriptionType(SubscriptionType.Shared)//仅共享消费模式支持重试和死信
    .enableRetry(true)
    .subscriptionName("sub1")
    .subscribe();
```

此时， `topic1` 对 `sub1` 的订阅就形成了带有重试机制的投递模式， `sub1` 会自动订阅之前在新建订阅时自动创建的重试 Topic 中（可以在控制台 Topic 列表中找到）。当 `topic1` 中的消息投递第一次未收到消费端 ACK 时，这条消息就会被自动投递到重试 Topic ，并且由于 consumer 自动订阅了这个主题，后续这条消息会在一定的 重试规则 下重新被消费。当达到最大重试次数后仍失败，消息会被投递到对应的死信队列，等待人工处理。

> **说明**
> 如果是 client 端自动创建的订阅，可以通过控制台上的 Topic 管理 > **更多** > **查看订阅**进入消费管理页面手动重建重试和死信队列。

自定义参数设置

TDMQ Pulsar 版会默认配置一套重试和死信参数，具体如下：

2.9.2 版本集群

指定重试次数为16次（失败16次后，第17次会投递到死信队列）
指定重试队列为 `[Topic 名称]-[订阅名]-RETRY`
指定死信队列为 `[Topic 名称]-[订阅名]-DLQ`

2.7.2 版本集群

指定重试次数为16次（失败16次后，第17次会投递到死信队列）
指定重试队列为 `[订阅名]-RETRY`
指定死信队列为 `[订阅名]-DLQ`

2.6.1 版本集群

指定重试次数为16次（失败16次后，第17次会投递到死信队列）
指定重试队列为 `[订阅名]-retry`
指定死信队列为 `[订阅名]-dlq`

如果希望自定义配置这些参数，可以使用 `deadLetterPolicy` API 进行配置，代码如下：

```
Consumer<byte[]> consumer = pulsarClient.newConsumer()
    .topic("persistent://pulsar-****")
    .subscriptionName("sub1")
    .subscriptionType(SubscriptionType.Shared)
    .enableRetry(true)//开启重试消费
    .deadLetterPolicy(DeadLetterPolicy.builder()
        .maxRedeliverCount(maxRedeliveryCount)//可以指定最大重试次数
        .retryLetterTopic("persistent://my-property/my-ns/sub1-retry")//可以指定重试队列
        .deadLetterTopic("persistent://my-property/my-ns/sub1-dlq")//可以指定死信队列
        .build())
    .subscribe();
```

重试规则

重试规则由 `reconsumerLater` API 实现，有三种模式：

```
//指定任意延迟时间
consumer.reconsumeLater(msg, 1000L, TimeUnit.MILLISECONDS);
//指定延迟等级
consumer.reconsumeLater(msg, 1);
//等级递增
consumer.reconsumeLater(msg);
```

- **第一种：指定任意延迟时间**。第二个参数填写延迟时间，第三个参数指定时间单位。延迟时间和延时消息的取值范围一致，范围在1 – 864000（单位：秒）。
- **第二种：指定任意延迟等级（仅限存量腾讯云版SDK的用户使用）**。实现效果和第一种基本一致，更方便统一管理分布式系统中的延时时长，延迟等级说明如下：

  1.1 `reconsumeLater(msg, 1)` 中的第二个参数即为消息等级。

  1.2 默认

  `MESSAGE_DELAYLEVEL = "1s 5s 10s 30s 1m 2m 3m 4m 5m 6m 7m 8m 9m 10m 20m 30m 1h 2h"`

  ，这个常数决定了每级对应的延时时间，例如1级对应1s，3级对应10s。如果默认值不符合实际业务需求，用户可以重新自定义。
- **第三种：等级递增（仅限存量腾讯云版 SDK 的用户使用）**。实现的效果不同于以上两种，为退避式的重试，即第一次失败后重试间隔为1秒，第二次失败后重试间隔为5秒，以此类推，次数越多，间隔时间越长。具体时间间隔同样由第二种中介绍的 `MESSAGE_DELAYLEVEL` 决定。
  这种重试机制往往在业务场景中有更实际的应用，如果消费失败，一般服务不会立刻恢复，使用这种渐进式的重试方式更为合理。

> **注意**

> 如果您使用的是 Pulsar 社区的 SDK，则不支持延迟等级和等级递增两种模式。

重试消息的消息属性

一条重试消息会给消息带上如下 property。

```
{
 REAL_TOPIC="persistent://my-property/my-ns/test,
 ORIGIN_MESSAGE_ID=314:28:-1,
 RETRY_TOPIC="persistent://my-property/my-ns/my-subscription-retry,
 RECONSUMETIMES=16
}
```

- `REAL_TOPIC`：原 Topic。
- `ORIGIN_MESSAGE_ID`：最初生产的消息 ID。
- `RETRY_TOPIC`：重试 Topic。
- `RECONSUMETIMES`：代表该消息重试的次数。

重试消息的消息 ID 流转

消息 ID 流转过程如下所示，您可以借助此规则对相关日志进行分析。

```
原始消费： msgid=1:1:0:1
第一次重试： msgid=2:1:-1
第二次重试： msgid=2:2:-1
第三次重试： msgid=2:3:-1
.......
第16次重试： msgid=2:16:0:1
第17次写入死信队列： msgid=3:1:-1
```

完整代码示例

- 重试（-RETRY）Topic 需要在 Consumer 中首先开启该功能（enableRetry(true)），默认为关闭状态。之后需要调用 reconsumeLater() 的接口消息才会被发送到重试 Topic 中。
- 死信（-DLQ）Topic 需要调用 consumer.reconsumeLater()，执行 reconsumeLater 之后原 topic 的那条消息会被 ack，消息转存到 retry topic，重试到达上限后消息转存至死信。Pulsar Client 会自动订阅 retry topic，但是进入死信就不会自动订阅，需要用户自己来订阅。

以下为借助 TDMQ Pulsar 版实现完整消息重试机制的代码示例，供开发者参考。

订阅主题

```
Consumer<byte[]> consumer1 = client.newConsumer()
```

```
        .topic("persistent://pulsar-****")
        .subscriptionName("my-subscription")
        .subscriptionType(SubscriptionType.Shared)
        .enableRetry(true)//开启重试消费
        //.deadLetterPolicy(DeadLetterPolicy.builder()
        //        .maxRedeliverCount(maxRedeliveryCount)
        //        .retryLetterTopic("persistent://my-property/my-ns/my-subscription-retry")//
可以指定重试队列
        //        .deadLetterTopic("persistent://my-property/my-ns/my-subscription-dlq")//
可以指定死信队列
        //        .build())
        .subscribe();
```

执行消费

```
while (true) {
    Message msg = consumer.receive();
    try {
        // Do something with the message
        System.out.printf("Message received: %s", new String(msg.getData()));
        // Acknowledge the message so that it can be deleted by the message broker
        consumer.acknowledge(msg);
    } catch (Exception e) {
        // select reconsume policy
        consumer.reconsumeLater(msg, 1000L, TimeUnit.MILLISECONDS);
        //consumer.reconsumeLater(msg, 1);
        //consumer.reconsumeLater(msg);
    }
}
```

主动重试

当消费者在某个时间没有成功消费某条消息，如果想重新消费到这条消息时，消费者可以发送一条取消确认消息到 TDMQ Pulsar 版服务端，TDMQ Pulsar 版会将这条消息重新发给消费者。 这种方式重试时不会产生新的消息，所以也不能自定义重试间隔。
以下为主动重试的 Java 代码示例：

```
while (true) {
   Message msg = consumer.receive();
   try {
      // Do something with the message
      System.out.printf("Message received: %s", new String(msg.getData()));
      // Acknowledge the message so that it can be deleted by the message broker
```

```
        consumer.acknowledge(msg);
    } catch (Exception e) {
        // Message failed to process, redeliver later
        consumer.negativeAcknowledge(msg);
    }
}
```

客户端连接与生产消费者

最近更新时间：2023-01-03 14:46:03

本文主要介绍 TDMQ Pulsar 客户端与连接、客户端与生产/消费者之间的关系，并向开发者介绍客户端合理的使用方式，以便更高效、稳定地使用 TDMQ Pulsar 版的服务。

> **核心原则：**
> - 一个进程一个 PulsarClient 即可。
> - Producer、Consumer 是线程安全的，对于同一个 Topic，可以复用且最好复用。

客户端与连接

TDMQ Pulsar 客户端（以下简称 PulsarClient ）是应用程序连接到 TDMQ Pulsar 版的一个基本单位，一个 PulsarClient 对应一个 TCP 连接。一般来说，用户侧的一个应用程序或者进程对应使用一个 PulsarClient，有多少个应用节点，对应就有多少个 Client 数量。若长时间不使用 TDMQ Pulsar 版服务的应用节点，应回收 Client 以节省资源消耗（当前 TDMQ Pulsar 版的连接上限是单个 Topic 200个 Client 连接）。

> **说明：**
> 如果业务侧 Topic 数量较多，确实需要创建多个 Client 可以使用如下方式复用 Client 对象：
> 1. 同一个 Topic 的多个 Producer 或者 Consumer 分别复用同一个 Client 对象
> 2. 如果【1】仍不满足，可以尝试多个 Topic 复用同一个 Client 对象
> 3. 此限制只针对共享集群，在专业集群中，默认配置仍为 200，但可以根据用户的实际需求调整该限制。

客户端与生产/消费者

一个 Client 下可以创建多个生产和消费者，用于提升生产和消费的速度。比较常见的用法是，一个 Client 下，利用多线程创建多个 Producer 或 Consumer 对象，用于生产消费，不同 Producer 和 Consumer 之间数据相互隔离。
当前 TDMQ Pulsar 版对生产/消费者的限制为：
- 单个 Topic 生产者上限1000个。
- 单个 Topic 消费者上限2000个。

最佳实践

生产/消费者的数量不一定取决于业务对象，它们是一个可以复用的资源，通过名称作为唯一标识进行区分。

生产者

假设有1000个业务对象在同时生产消息，并不是要创建1000个 Producer，只要是向同一个 Topic 进行投递，每个应用节点可以先统一使用一个 Producer 来进行生产（单例模式），往往单个 Producer 就能吃满单个应用节点的硬件配置。
以下给出一段 Java 消息生产的代码示例。

```
//从配置文件中获取 serviceURL 接入地址、Token 密钥、Topic 全名和 Subscription 名称（均可从控制台复制）
@Value("${tdmq.serviceUrl}")
private String serviceUrl;
@Value("${tdmq.token}")
private String token;
@Value("${tdmq.topic}")
private String topic;

//声明1个 Client 对象、producer 对象
private PulsarClient pulsarClient;
private Producer<String> producer;

//在一段初始化程序中创建好客户端和生产者对象
public void init() throws Exception {
    pulsarClient = PulsarClient.builder()
            .serviceUrl(serviceUrl)
            .authentication(AuthenticationFactory.token(token))
            .build();
    producer = pulsarClient.newProducer(Schema.STRING)
            .topic(topic)
            .create();
}
```

在实际生产消息的业务逻辑中直接引用 `producer` 完成消息的发送。

```
//在实际生产消息的业务逻辑中直接引用，注意 Producer 通过范式声明的 Schema 类型要和传入对象匹配
public void onProduce(Producer<String> producer){
    //添加业务逻辑
    String msg = "my-message";//模拟从业务逻辑拿到消息
    try {
        //TDMQ Pulsar 版默认开启 Schema 校验, 消息对象一定需要和 producer 声明的 Schema 类型匹配
        MessageId messageId = producer.newMessage()
            .key("msgKey")
                .value(msg)
                .send();
        System.out.println("delivered msg " + msgId + ", value:" + value);
```

```
    } catch (PulsarClientException e) {
        System.out.println("delivered msg failed, value:" + value);
        e.printStackTrace();
    }
}

public void onProduceAsync(Producer<String> producer){
    //添加业务逻辑
    String msg = "my-asnyc-message";//模拟从业务逻辑拿到消息
    //异步发送消息，无线程阻塞，提升发送速率
    CompletableFuture<MessageId> messageIdFuture = producer.newMessage()
        .key("msgKey")
            .value(msg)
            .sendAsync();
    //通过异步回调得知消息发送成功与否
    messageIdFuture.whenComplete(((messageId, throwable) -> {
        if( null != throwable ) {
            System.out.println("delivery failed, value: " + msg );
            //此处可以添加延时重试的逻辑
        } else {
            System.out.println("delivered msg " + messageId + ", value:" + msg);
        }
    }));
}
```

当一个生产者长时间不使用时需要调用 close 方法关闭，以避免占用资源；当一个客户端实例长时间不使用时，同样需要调用 close 方法关闭，以避免连接池被占满。

```
public void destroy(){
    if (producer != null) {
        producer.close();
    }
    if (pulsarClient != null) {
        pulsarClient.close();
    }
}
```

消费者

如同生产者，消费者也最好按照单例模式进行使用，单个消费节点只需要一个客户端实例以及一个消费者实例。一般来说，一个消息队列的消费端的性能瓶颈都在于消费者按照自己业务逻辑处理消息的过程，而并非在接收消息的动作上。所以当出现了消费性能不足的时候，先看消费者的网络带宽消耗，如果趋势上看没有达到一个明显的上限，就应该先根据日志以及消息轨迹信息分析自身处理消息的业务逻辑耗时。

注意

- 当使用 Shared 或者 Key-Shared 模式时，消费者数量不一定小于等于分区数。服务端会有一个负责分发消息的模块按照一定的方式（Shared 模式默认是轮询，Key-Shared 则是在同一个 key 内轮询）将消息分发给所有的消费者。
- 当使用 Shared模式，如果生产侧暂停了生产，则到了末尾一部分消息时，可能会出现消费分布不均的情况。
- 使用多线程消费，即使复用一个 consumer 对象，消息的顺序也将无法得到保证。

以下给出一个 Java 基于 Spring boot 框架用线程池进行多线程消费的完整代码示例。

```
import org.apache.pulsar.client.api.*;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Service;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;

import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

@Service
public class ConsumerService implements Runnable {

    //从配置文件中获取 serviceURL 接入地址、Token 密钥、Topic 全名和 Subscription 名称
（均可从控制台复制）
    @Value("${tdmq.serviceUrl}")
    private String serviceUrl;
    @Value("${tdmq.token}")
    private String token;
    @Value("${tdmq.topic}")
    private String topic;
    @Value("${tdmq.subscription}")
    private String subscription;

    private volatile boolean start = false;
    private PulsarClient pulsarClient;
    private Consumer<String> consumer;
    private static final int corePoolSize = 10;
    private static final int maximumPoolSize = 10;
```

```
    private ExecutorService executor;
    private static final Logger logger =
LoggerFactory.getLogger(ConsumerService.class);

    @PostConstruct
    public void init() throws Exception {
        pulsarClient = PulsarClient.builder()
                .serviceUrl(serviceUrl)
                .authentication(AuthenticationFactory.token(token))
                .build();
        consumer = pulsarClient.newConsumer(Schema.STRING)
                .topic(topic)
                //.subscriptionType(SubscriptionType.Shared)
                .subscriptionName(subscription)
                .subscribe();
        executor = new ThreadPoolExecutor(corePoolSize, maximumPoolSize, 0,
TimeUnit.SECONDS, new ArrayBlockingQueue<>(100),
                new ThreadPoolExecutor.AbortPolicy());
        start = true;
    }

    @PreDestroy
    public void destroy() throws Exception {
        start = false;
        if (consumer != null) {
            consumer.close();
        }
        if (pulsarClient != null) {
            pulsarClient.close();
        }
        if (executor != null) {
            executor.shutdownNow();
        }
    }

    @Override
    public void run() {
        logger.info("tdmq consumer started...");
        for (int i = 0; i < maximumPoolSize; i++) {
            executor.submit(() -> {
                while (start) {
                    try {
                        Message<String> message = consumer.receive();
                        if (message == null) {
```

```
                    continue;
                }
                onConsumer(message);
            } catch (Exception e) {
                logger.warn("tdmq consumer business error", e);
            }
        }
    });
    }
    logger.info("tdmq consumer stopped...");
  }

  /**
   * 这里写消费业务逻辑
   *
   * @param message
   * @return return true: 消息ack  return false: 消息nack
   * @throws Exception 消息nack
   */
  private void onConsumer(Message<String> message) {
    //业务逻辑,延时类操作
    try {
      System.out.println(Thread.currentThread().getName() + " - message receive: "
+ message.getValue());
      Thread.sleep(1000);//模拟业务逻辑处理
      consumer.acknowledge(message);
      logger.info(Thread.currentThread().getName() + " - message processing
succeed: " + message.getValue());
    } catch (Exception exception) {
      consumer.negativeAcknowledge(message);
      logger.error(Thread.currentThread().getName() + " - message processing
failed: " + message.getValue());
    }
  }
}
```