

Apache Pulsar 高级篇

云原生时代消息中间件之王





目录

Contents

- ◆ Pulsar高级组件基本使用 (Connector, Functions, 事务)
- ◆ Pulsar架构原理 (Broker与Bookkeeper)
- ◆ Pulsar的读写流程
- ◆ Pulsar的安全机制
- ◆ Pulsar的跨机房复制
- ◆ Pulsar的KOP与AOP

学习目标

Learning Objectives

1. 掌握Pulsar的高级组件使用
2. 理解Pulsar的相关核心原理
3. 了解Pulsar的安全机制
4. 理解并知道Pulsar如何完成跨机房复制
5. 完成Pulsar的KOP与AOP



01

Pulsar高级组件基本使用

- Funcation(轻量级计算流程)概念与使用
- Connector 连接器概念与使用
- Transactions事务支持相关的操作

Pulsar Function 轻量级计算框架

- Function背景介绍

当我们进行流式处理的时候，很多情况下，我们的需求可能只是下面这些简单的操作：简单的 ETL 操作\聚合计算操作等相关服务。

但为了实现这些功能，我们不得不去部署一整套 SPE 服务。部署成功后才发现需要的仅是 SPE (流处理引擎) 服务中的一小部分功能，部署 SPE 的成本可能比用户开发这个功能本身更困难。由于 SPE 本身 API 的复杂性，我们需要了解这些算子的使用场景，明白不同算子之间有哪些区别，什么情况下，应该使用什么算子来处理相应的逻辑。

基于以上原因，我们设计并实现了 Pulsar Functions，在 Pulsar Functions 中，用户只需关心计算逻辑本身，而不需要去了解或者部署 SPE 的相关服务，当然你也可以将 pulsar-function 与现有的 SPE 服务^[1]一起使用。也就是说，在 Pulsar Functions 中，无需部署 SPE 的整套服务，就可以达到与 SPE 服务同样的优势。

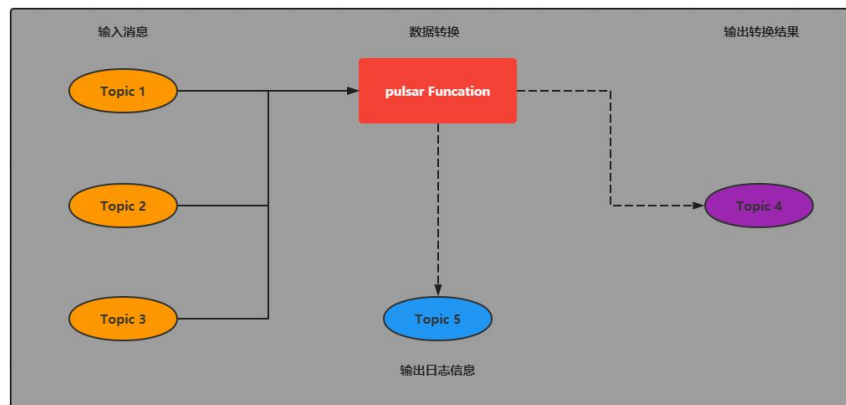
Pulsar Function 轻量级计算框架

- 什么是Functions

Pulsar Functions 是一个轻量级的计算框架，像 AWS 的 lambda、Google Cloud 的 Functions 一样，Pulsar Functions 可以给用户提供一个部署简单、运维简单、API 简单的 FASS (Function as a service) 平台。

Pulsar Functions 的设计灵感来自于 Heron 这样的流处理引擎，Pulsar Functions 将会拓展 Pulsar 和整个消息领域的未来。使用 Pulsar Functions，用户可以轻松地部署和管理 function，通过 function 从 Pulsar topic 读取数据或者生产新数据到 Pulsar topic。

引入 Pulsar Functions 后，Pulsar 成为统一的消息投递/计算/存储平台。只需部署一套 Pulsar 集群，便可以实现一个计算引擎，页面简单，操作便捷。



Pulsar Function 轻量级计算框架

- 什么是Functions

Input topic 是数据的来源，在 Pulsar Functions 中，所有的数据均来自 input topic。当数据进入input topic 中，Pulsar Functions 充当消费者的角色，去 input topic 中消费消息；当从 input topic 中拿到需要处理的消息时，Pulsar Functions 充当生产者的角色往 output topic 或者 log topic 中生产消息。

Output topic 和 log topic 都可以看作是 Pulsar Functions 的输出。从是否会有 output 这个点来看，我们可以将 Pulsar Functions 分为两类，当有输出的时候 Pulsar Functions 会将相应的 output 输出到 output topic 中。log topic 主要存储用户的日志信息，当 Pulsar Functions 出现问题时，方便用户定位错误并调试。

综上所述：我们不难看出 Pulsar Functions 充当了一个消息处理和转运的角色。

在使用Pulsar Functions，可以使用不同的语言来编写，比如Python, Java, Go等。编写方式主要两种

- 本地模式：集群外部，进行本地运行
- 集群模式：集群内部运行(支持独立模式和集成模式)

Pulsar Function 轻量级计算框架

如何使用呢?

首先，需要修改Pulsar中相关的配置：

```
cd /export/server/pulsar_2.8.1/conf  
vim broker.conf
```

内容如下：

修改1161行：

functionsWorkerEnabled=false 更改为 true

注意：三台节点都需要调整

接着，重启Broker即可

```
cd /export/server/pulsar_2.8.1  
bin/pulsar-daemon stop broker  
bin/pulsar-daemon start broker
```

注意：三台节点都需要执行,依次都停止,然后依次启动

Pulsar Function 轻量级计算框架

如何使用呢?

最后, 测试是否可用

```
cd /export/server/pulsar_2.8.1/  
bin/pulsar-admin functions create \  
--jar examples/api-examples.jar \  
--classname org.apache.pulsar.functions.api.examples.ExclamationFunction \  
--inputs persistent://public/default/exclamation-input \  
--output persistent://public/default/exclamation-output \  
--tenant public \  
--namespace default \  
--name exclamation
```

```
"Created successfully"
```

检查是否按照预期触发函数运行:

```
bin/pulsar-admin functions trigger --name exclamation --trigger-value "hello world"
```

```
[root@node1 pulsar_2.8.1]# bin/pulsar-admin functions trigger --name exclamation --trigger-value "hello world"  
hello world!
```

Pulsar Function 轻量级计算框架

如何使用呢?

bin/pulsar-admin functions

属性说明:

functions:

可选值:

localrun: 创建本地function进行运行

create: 在集群模式下创建

delete: 删除在集群中运行的function

get: 获取function的相关信息

restart: 重启

stop : 停止运行

start: 启动

status: 检查状态

stats: 查看状态

list: 查看特定租户和名称空间下的所有的function

--classname: 设置function执行类

--jar 设置function对应的jar包

--inputs : 输入的topic

--output : 输出的topic

--tenant : 设置function运行在那个租户中

--namespace: 设置function运行在那个名称空间中

--name : 定义function的名称

Pulsar Function 轻量级计算框架

接下来, 我们尝试编写一个function的操作, 基于Pulsar Function完成流式计算操作:

案例需求:

使用Pulsar Function 读取某一个Topic中日期(格式为: yyyy/MM/dd HH/mm/ss)数据, 读取后, 对数据进行日期转换(格式为:yyyy-MM-dd HH:mm:ss)

首先加入依赖:

```
<dependency>
  <groupId>org.apache.pulsar</groupId>
  <artifactId>pulsar-functions-api</artifactId>
  <version>2.6.0</version>
</dependency>
```

接着编写程序

Pulsar Function 轻量级计算框架

接着编写程序

```
package com.itheima.functions;

import org.apache.pulsar.functions.api.Context;
import org.apache.pulsar.functions.api.Function;

import java.text.SimpleDateFormat;
import java.util.Date;

public class WordCountFunction implements Function<String,String> {
    private SimpleDateFormat format1 = new SimpleDateFormat("yyyy/MM/dd HH/mm/ss");
    private SimpleDateFormat format2 = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

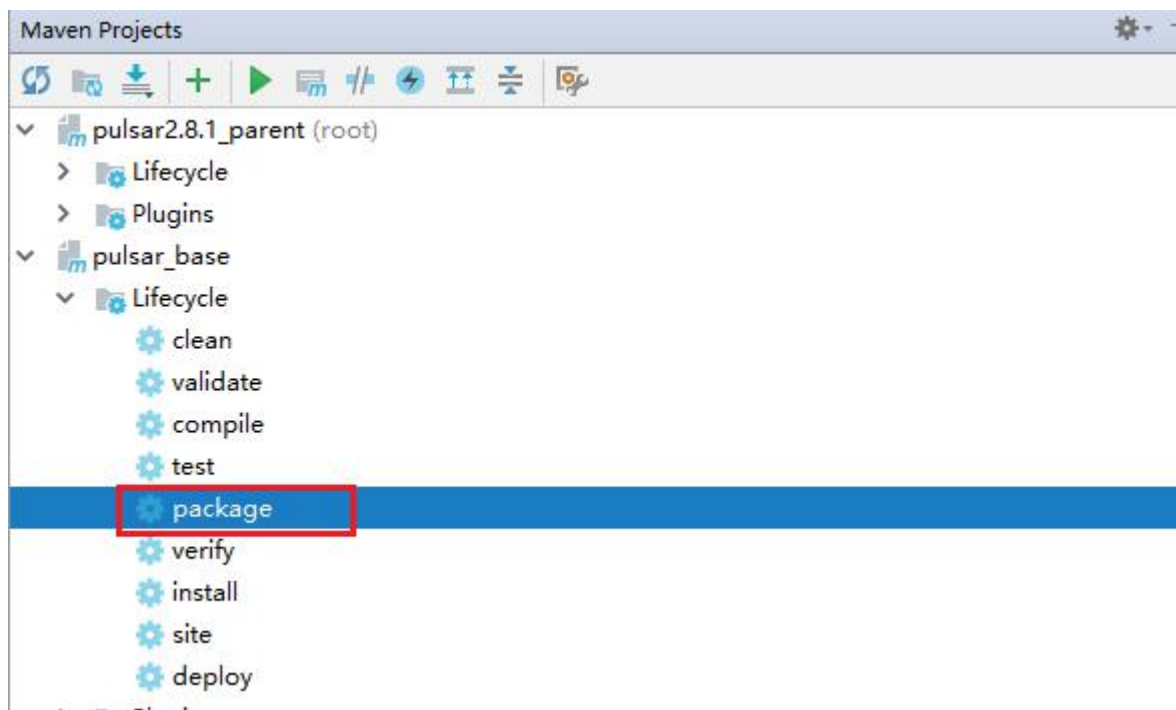
    @Override
    public String process(String input, Context context) throws Exception {

        Date oldDate = format1.parse(input);

        return format2.format(oldDate);
    }
}
```

Pulsar Function 轻量级计算框架

对项目程序进行打包部署



打包后, 在target下会出现一个Jar包, 将其上传到服务器(路径无所谓)



Pulsar Function 轻量级计算框架

构建function

```
cd /export/server/pulsar_2.8.1/  
bin/pulsar-admin functions create \  
--jar functions/pulsar_base-1.0-SNAPSHOT.jar \  
--classname com.itheima.functions.WordCountFunction \  
--inputs persistent://public/default/wd_input \  
--output persistent://public/default/wd_output \  
--tenant public\  
--namespace default \  
--name wordcount
```

```
[root@node1 pulsar_2.8.1]# bin/pulsar-admin functions create --jar functions/pulsar_base-1.0-SNAPSHOT.jar --classname  
com.itheima.functions.WordCountFunction --inputs persistent://public/default/wd_input --output persistent://public/  
default/wd_output --tenant public --namespace default --name wordcount  
"Created successfully"
```

Pulsar Function 轻量级计算框架

启动function

trigger 触发启动, 并向函数发送数据测试

```
bin/pulsar-admin functions trigger --name wordcount --trigger-value "2021/10/10 15/30/30"
```

```
[root@node1 pulsar_2.8.1]# bin/pulsar-admin functions trigger --name wordcount --trigger-value "2021/10/10 15/30/30"  
2021-10-10 15:30:30
```

此外, 大家也可以通过代码向input对应的Topic发送消息, 并消费output对应的Topic中数据, 也是可以看到function可以正常处理的



01

Pulsar高级组件基本使用

- Function(轻量级计算流程)概念与使用
- **Connector 连接器概念与使用**
- Transactions事务支持相关的操作

Pulsar Connector 连接器

虽然可以使用 Pulsar 消费者和生产者 API 编写代码（例如，从数据库同步数据时，先查询数据，再使用 Pulsar 的 API 将数据发布至 Pulsar），但这种方法耗时费力。因此，Pulsar 提出了 Connector（也称为 Pulsar IO），用于解决 Pulsar 与周边系统的集成问题，帮助用户高效完成工作。



这张图非常直观地描述了 Pulsar IO 的组成。

Pulsar IO 分为输入（Input）和输出（Output）两个模块。

输入：代表数据从哪里来，通过 Source 实现数据输入。数据的来源可以是数据库（例如 MySQL、Oracle、MongoDB）、文件、日志或自定义系统等。

输出：代表数据往哪里去，通过 Sink 实现数据输出。数据的输出可以是数据仓库、数据库等。

Pulsar Connector 连接器

而目前Pulsar支持非常多的Connector. 有兴趣的可以参考以下几个网站：

<http://pulsar.apache.org/docs/zh-CN/io-connectors/#source-connector>

<http://pulsar.apache.org/docs/zh-CN/io-connectors/#sink-connector>

目前我们主要介绍 [Pulsar flink Connector](#) 和 [Pulsar Flume Connector](#)

其他的连接器的使用方式，基本是类似的

Source connector

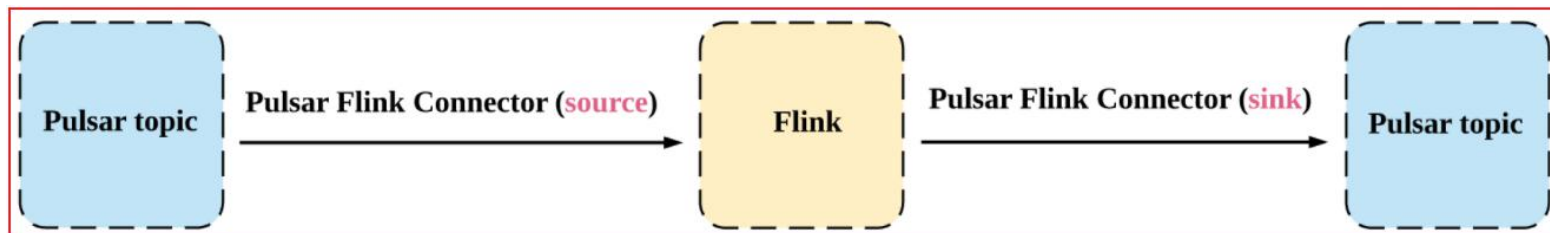
Canal
Debezium MySQL
Debezium PostgreSQL
Debezium MongoDB
DynamoDB
File
Flume
Twitter firehose
Kafka
Kinesis
Netty
NSQ
RabbitMQ

Sink connector

Aerospike
Cassandra
ElasticSearch
Flume
HBase
HDFS2
HDFS3
InfluxDB
JDBC ClickHouse
JDBC MariaDB
JDBC PostgreSQL
JDBC SQLite
Kafka
Kinesis
MongoDB
RabbitMQ
Redis
Solr

Pulsar Connector 连接器 ----> Pulsar Flink Connector

Pulsar Flink Connector是Apache Pulsar和Apache Flink(数据处理引擎)的集成，它允许Flink从Pulsar读取数据，并向Pulsar写入数据，并提供精确一次的源语义和至少一次的汇聚语义。



如何使用pulsar Flink Connector，首先在pom中加入相关的依赖环境:(注意：还需要添加 pulsar客户端包)

```
<repositories><!--代码库-->
<repository>
  <id>aliyun</id>
  <url>http://maven.aliyun.com/nexus/content/groups/public/</url>
  <releases><enabled>true</enabled></releases>
  <snapshots>
    <enabled>false</enabled>
    <updatePolicy>never</updatePolicy>
  </snapshots>
</repository>
</repositories>
```

Pulsar Connector 连接器 ----> Pulsar Flink Connector

```
<dependencies>
  <dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-java</artifactId>
    <version>1.13.1</version>
  </dependency>
  <dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-streaming-java_2.11</artifactId>
    <version>1.13.1</version>
  </dependency>

  <dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-clients_2.11</artifactId>
    <version>1.13.1</version>
  </dependency>

  <dependency>
    <groupId>org.apache.pulsar</groupId>
    <artifactId>pulsar-client-all</artifactId>
    <version>2.8.1</version>
  </dependency>
```

Pulsar Connector 连接器 ----> Pulsar Flink Connector

```
<dependency>
  <groupId>io.streamnative.connectors</groupId>
  <artifactId>pulsar-flink-connector_2.11</artifactId>
  <version>1.13.1.5-rc1</version>
  <exclusions>
    <exclusion>
      <groupId>org.apache.pulsar</groupId>
      <artifactId>pulsar-client-all</artifactId>
    </exclusion>
  </exclusions>
</dependency>
</dependencies>
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.1</version>
      <configuration>
        <target>1.8</target>
        <source>1.8</source>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Pulsar Connector 连接器 ----> Pulsar Flink Connector

如何在flink的流式环境中使用Pulsar: source端

```
public class FlinkFromPulsarSource {
    public static void main(String[] args) throws Exception {
        // 1) 创建flink的流式计算核心环境类对象 StreamExecutionEnvironment.getExecutionEnvironment();
        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
        // 2) 添加source数据源, 用于读取数据
        Properties props = new Properties();
        props.setProperty("topic", "persistent://itcast_pulsar_t/itcast_pulsar_n/my-topic3");
        props.setProperty("partition.discovery.interval-millis", "5000");
        FlinkPulsarSource<String> pulsarSource = new FlinkPulsarSource<String>(
            "pulsar://node1:6650,node2:6650,node3:6650", "http://node1:8080,node2:8080,node3:8080",
            PulsarDeserializationSchema.valueOnly(new SimpleStringSchema()), props);

        pulsarSource.setStartFromLatest();
        DataStreamSource<String> source = env.addSource(pulsarSource);
        // 3) 添加相关的转换操作, 对数据进行分析处理
        // 4) 添加sink组件, 将计算后结果进行输出操作
        source.print();
        // 5) 启动flink程序
        env.execute("FlinkFromPulsar");
    }
}
```

Pulsar Connector 连接器 ----> Pulsar Flink Connector

如何在flink的流式环境中使用Pulsar: source消费, sink生产到Pulsar中

```
public class FlinkFromPulsarSink {
    public static void main(String[] args) throws Exception {
        // 1) 创建flink的流式计算核心环境类对象 StreamExecutionEnvironment.getExecutionEnvironment();
        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
        // 2) 添加source数据源, 用于读取数据
        Properties props = new Properties();
        props.setProperty("topic", "persistent://itcast_pulsar_t/itcast_pulsar_n/my-topic3");
        props.setProperty("partition.discovery.interval-millis", "5000");
        FlinkPulsarSource<String> pulsarSource = new FlinkPulsarSource<String>(
            "pulsar://node1:6650,node2:6650,node3:6650", "http://node1:8080,node2:8080,node3:8080",
            PulsarDeserializationSchema.valueOnly(new SimpleStringSchema()), props);
        pulsarSource.setStartFromLatest();
        DataStreamSource<String> source = env.addSource(pulsarSource);
        // 3) 添加相关的转换操作, 对数据进行分析处理
        // 4) 添加sink组件, 将计算后结果进行输出到Pulsar
        /* 基于pojo的方式*/
        /*PulsarSerializationSchema<Person> pulsarSerialization = new
        PulsarSerializationSchemaWrapper.Builder<>(JsonSer.of(Person.class))
            .usePojoMode(Person.class, RecordSchemaType.JSON)
            .setTopicExtractor(person -> null)
            .build();*/
    }
}
```

Pulsar Connector 连接器 ----> Pulsar Flink Connector

如何在flink的流式环境中使用Pulsar: source消费, sink生产到Pulsar中

```
PulsarSerializationSchema<String> pulsarSerialization = new
PulsarSerializationSchemaWrapper.Builder<String>(JsonSer.of(String.class))
    .useAtomicMode(new AtomicDataType(new VarCharType()))
    .build();

FlinkPulsarSink<String> pulsarSink = new FlinkPulsarSink(
    "pulsar://node1:6650,node2:6650,node3:6650",
    "http://node1:8080,node2:8080,node3:8080",
    Optional.of("persistent://itcast_pulsar_t/itcast_pulsar_n/my-topic1"), // mandatory target topic or use
    `Optional.empty()` if sink to different topics for each record
    props,
    pulsarSerialization
);

source.addSink(pulsarSink);
// 5) 启动flink程序
env.execute("FlinkFromPulsar");
}
```


Pulsar Connector 连接器 ----> Pulsar Flume Connector

如何基于Flume完成采集数据到Pulsar:

- 第一步: 在windows下载相关源码

我的电脑 > 本地磁盘 (E:) > flume_source > flume-ng-pulsar-sink >

名称	修改日期	类型	大小
.git	2021/7/21 14:05	文件夹	
.github	2021/7/21 14:05	文件夹	
src	2021/7/21 14:05	文件夹	
.gitignore	2021/7/21 14:05	文本文档	1 KB
.travis.yml	2021/7/21 14:05	YML 文件	2 KB
dependency-reduced-pom.xml	2021/7/21 14:06	XML 文档	2 KB
LICENSE	2021/7/21 14:05	文件	12 KB
pom.xml	2021/7/21 14:05	XML 文档	3 KB
README.md	2021/7/21 14:05	Markdown File	8 KB

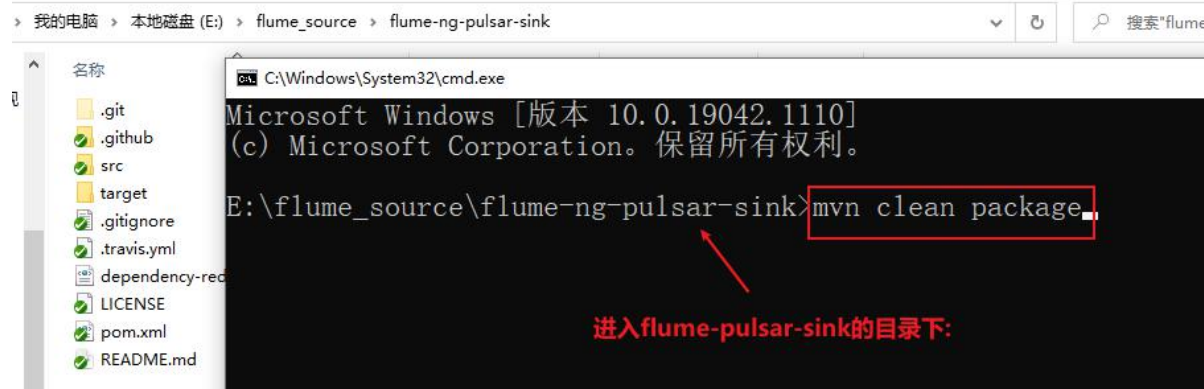
- 第二步: 执行打包命令
mvn clean package

Pulsar Connector 连接器 ----> Pulsar Flume Connector

如何基于Flume完成采集数据到Pulsar:

- 第二步: 执行打包命令

mvn clean package

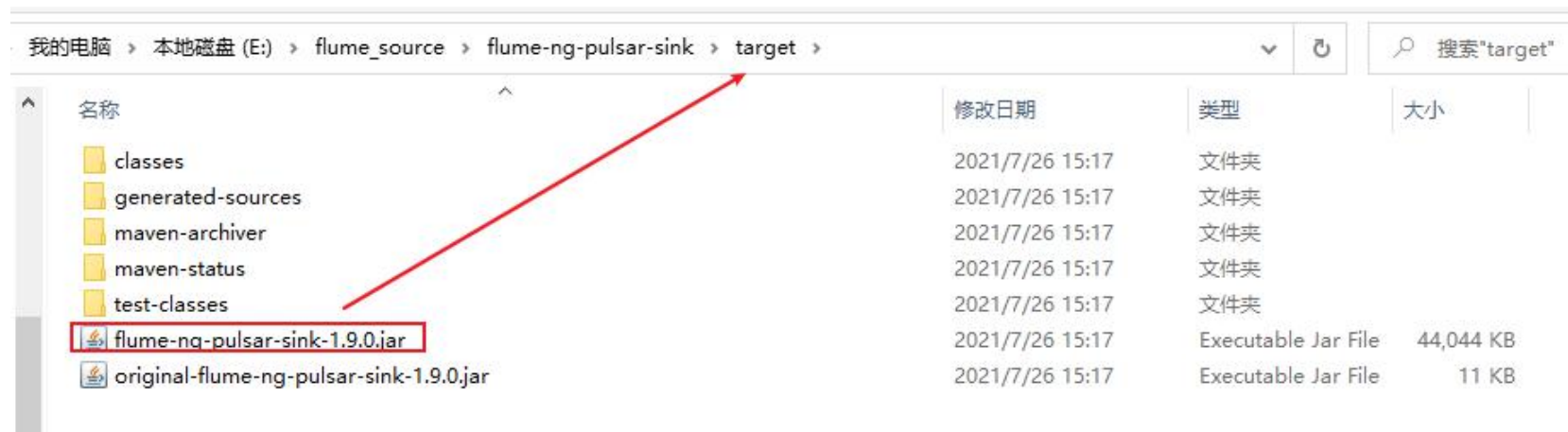


```
INFO] Dependency-reduced POM written at: E:\flume_source\flume-ng-pulsar-sink\dependency-reduced-pom.xml
INFO] -----
INFO] BUILD SUCCESS
INFO] -----
INFO] Total time: 8.637 s
INFO] Finished at: 2021-07-26T15:17:54+08:00
INFO] Final Memory: 38M/543M
INFO] -----
.: \flume_source\flume-ng-pulsar-sink>
```

Pulsar Connector 连接器 ----> Pulsar Flume Connector

如何基于Flume完成采集数据到Pulsar:

- 此时会产生一个target目录, 进入目录下, 就看到已经打好的包了:



- 第三步: 将jar上传到flume的lib目录下

在安装flume的服务器节点上:

```
cd${FLUME_HOME}/lib
```

```
rz 上传即可
```

Pulsar Connector 连接器 ----> Pulsar Flume Connector

如何基于Flume完成采集数据到Pulsar:

- 第四步: 配置Flume的采集文件

```
a1.sources = r1
a1.channels = c1
a1.sinks = k1

a1.sources.r1.type = netcat
a1.sources.r1.bind = node1
a1.sources.r1.port = 44444

a1.channels.c1.type = memory
a1.channels.c1.capacity = 100
a1.channels.c1.transactionCapacity = 100

a1.sinks.k1.serviceUrl = node1.itcast.cn:6650,node2.itcast.cn:6650,node3.itcast.cn:6650
a1.sinks.k1.topicName = flume-test-topic
a1.sinks.k1.producerName = flume-test-producer

a1.sources.r1.channels = c1
a1.sinks.k1.channel = c1
```

Pulsar Connector 连接器 ----> Pulsar Flume Connector

如何基于Flume完成采集数据到Pulsar:

- 第五步: 执行启动Flume

```
./flume-ng agent -n a1 -c ../conf/ -f ../conf/netcat_source_pulsar_sink.conf -Dflume.root.logger=INFO,console
```

- 第六步: 执行启动Flume

```
[root@node2 conf]# telnet node1 44444
Trying 192.168.88.161...
Connected to node1.
Escape character is '^]'.
11
OK
22
OK
33
OK
```

- 第七步: 观察消费者是否可以正常消费数据



01

Pulsar高级组件基本使用

- Function(轻量级计算流程)概念与使用
- Connector 连接器概念与使用
- Transactions事务支持相关的操作

Pulsar如何实现Exactly-Once

Apache Pulsar 社区在刚刚发布的 Pulsar 2.8.0 版本中实现了一个里程碑式功能：**Exactly-once**（精确一次）语义。在这之前，我们只能通过在 Broker 端开启消息去重来保证单个 Topic 上的 Exactly-once 语义。随着 Pulsar 2.8.0 的发布，利用事务 API 可以在跨 Topic 的场景下保证消息生产和确认的原子性操作

消息语义主要分为有至少一次，最多一次，精确一次 三种确认方案：

至少一次：

Producer 通过接收 Broker 的 ACK（消息确认）通知来确保消息成功写入 Pulsar Topic。然而，当 Producer 接收 ACK 通知超时，或者收到 Broker 出错信息时，会尝试重新发送消息。如果 Broker 正好在成功把消息写入到 Topic，但还没有给 Producer 发送 ACK 时宕机，Producer 重新发送的消息会被再次写入到 Topic，最终导致消息被重复分发至 Consumer。

最多一次：

当 Producer 在接收 ACK 超时，或者收到 Broker 出错信息时不重发消息，那就有可能导致这条消息丢失，没有写入到 Topic 中，也不会被 Consumer 消费到。在某些场景下，为了避免发生重复消费，我们可以容许消息丢失的发生。

精确一次：

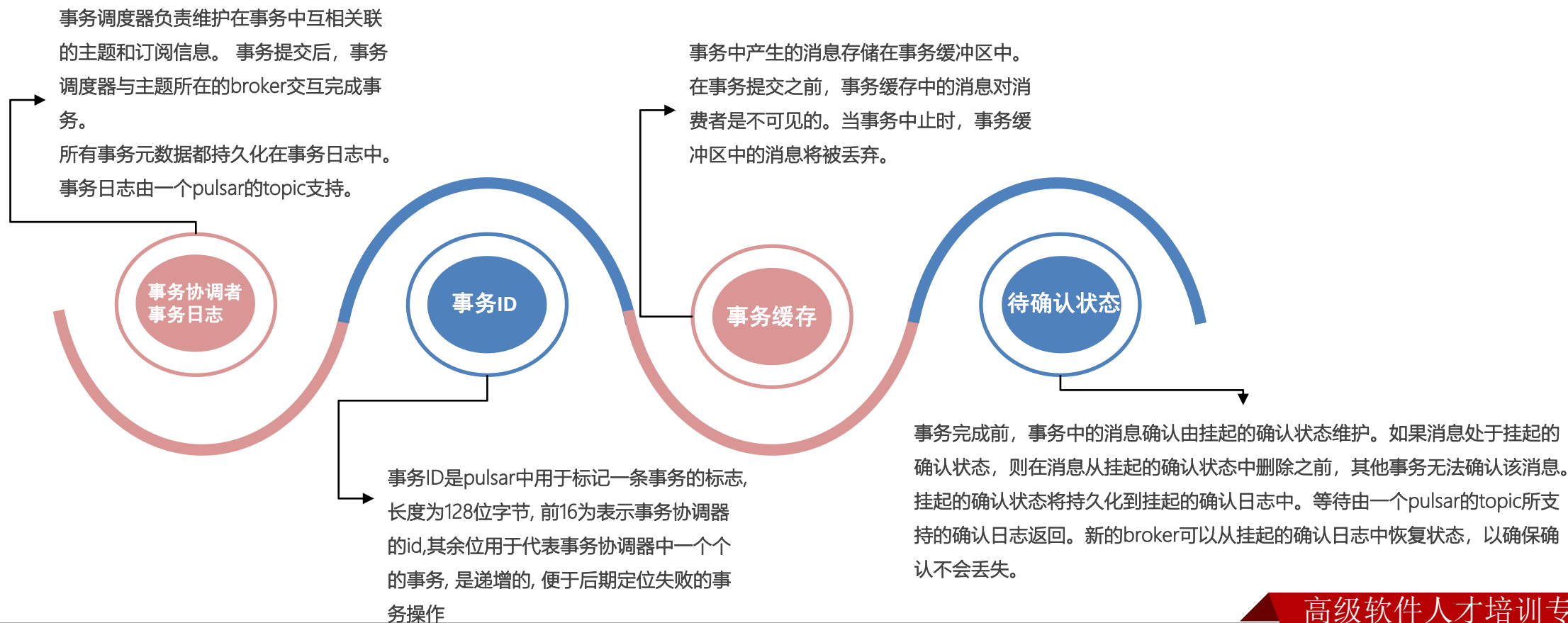
保证了即使 Producer 多次发送同一条消息到服务端，服务端也仅仅会记录一次。Exactly-once 语义是最可靠的，同时也是最难理解的。Exactly-once 语义需要消息队列服务端，消息生产端和消费端应用三者的协同才能实现。比如，当消费端应用成功消费并且 ACK 了一条消息之后，又把消费位点回滚到之前的一个消息 ID，那么从那个消息 ID 往后的所有消息都会被消费端应用重新消费到。

Pulsar如何实现Exactly-Once

Pulsar在最新的版本中，通过事务API 实现跨topic消息生产和确认的原子性操作，通过这个功能，Producer 可以确保一条消息同时发送到多个 Topic，要么这些消息都发送成功，在所有 Topic 上都可以被消费，要么所有消息都不能被消费。这个功能也允许在一个事务操作中对多个 Topic 上的消息进行 ACK 确认，从而实现端到端的 Exactly-once 语义。

Pulsar Transactions (事务支持)

事务语义允许事件流应用将消费，处理，生产消息整个过程定义为一个原子操作。在 Pulsar 中，生产者或消费者能够处理跨多个主题和分区的信息，允许一个原子操作写入多个主题和分区，同时事务中的批量消息可以被以多分区接收、生产和确认，一个事务涉及的所有操作都作为整体成功或失败。事务中几个概念需要大家了解：



Pulsar Transactions (事务支持)

- 第一步: 修改pulsar的broker.conf文件: 开启事务支持

#1257行: 开启事务支持

transactionCoordinatorEnabled=true

#468行: 开启批量确认

acknowledgmentAtBatchIndexLevelEnabled=true

- 第二步: 初始化事务协调器元数据

```
bin/pulsar initialize-transaction-coordinator-metadata -cs node1:2181,node2:2181,node3:2181 -c pulsar-cluster
```

```
20:20:43.521 [main-EventThread] INFO org.apache.pulsar.metadata.impl.ZKSessionWatcher - Got ZK session watch event: WatchedEvent state:Closed type:None path:null
20:20:43.526 [main] INFO org.apache.zookeeper.ZooKeeper - Session: 0x27dc5f090b2000e closed
Transaction coordinator metadata setup success
20:20:43.528 [main-EventThread] INFO org.apache.zookeeper.ClientCnxn - EventThread shut down for session: 0x27dc5f090b2000e
```

- 第三步: 在代码中构建一个支持事务的Pulsar的客户端

```
PulsarClient pulsarClient = PulsarClient.builder()
    .serviceUrl("pulsar://node1:6650,node2:6650,node3:6650")
    .enableTransaction(true)
    .build();
```

Pulsar Transactions (事务支持)

- 第四步: 开启事务支持

```
Transaction txn = pulsarClient.newTransaction().withTransactionTimeout(5, TimeUnit.MINUTES)
    .build().get();
```

- 第五步: 执行相关操作:

```
try {
    //3.1: 接收消息
    Consumer<byte[]> consumer = pulsarClient.newConsumer()
        .topic("persistent://public/default/txn_t1")
        .subscriptionName("my-subscription")
        //.enableBatchIndexAcknowledgment(true) 开启批量消息确认
        .subscribe();

    //3.2 获取消息
    Message<byte[]> message = consumer.receive();

    System.out.println("消息为:" + message.getTopicName() + ":" + new String(message.getData()));
}
```

Pulsar Transactions (事务支持)

```
//3.3 将接收到的消息, 处理后, 发送到另一个Topic中
Producer<byte[]> producer = pulsarClient.newProducer()
    .topic("persistent://public/default/txn_t2")
    .sendTimeout(0,TimeUnit.MILLISECONDS)
    .create();

producer.newMessage(txn).value(message.getData()).send();
System.out.println(1111);
//3.4: 确认输入的消息
consumer.acknowledge(message);
//4. 如果正常, 就提交事务
txn.commit();
}catch (Exception e){
    System.out.println(1111);
    // 否则就回滚事务
    txn.abort();
    e.printStackTrace();
}
}
```

在测试时, 可以通过在提交数据之前, 制造一个小错误, 让其抛出异常, 观察其是否可以发送成功, 并且在重新跑, 是否还会重新获取之前数据进行消费操作



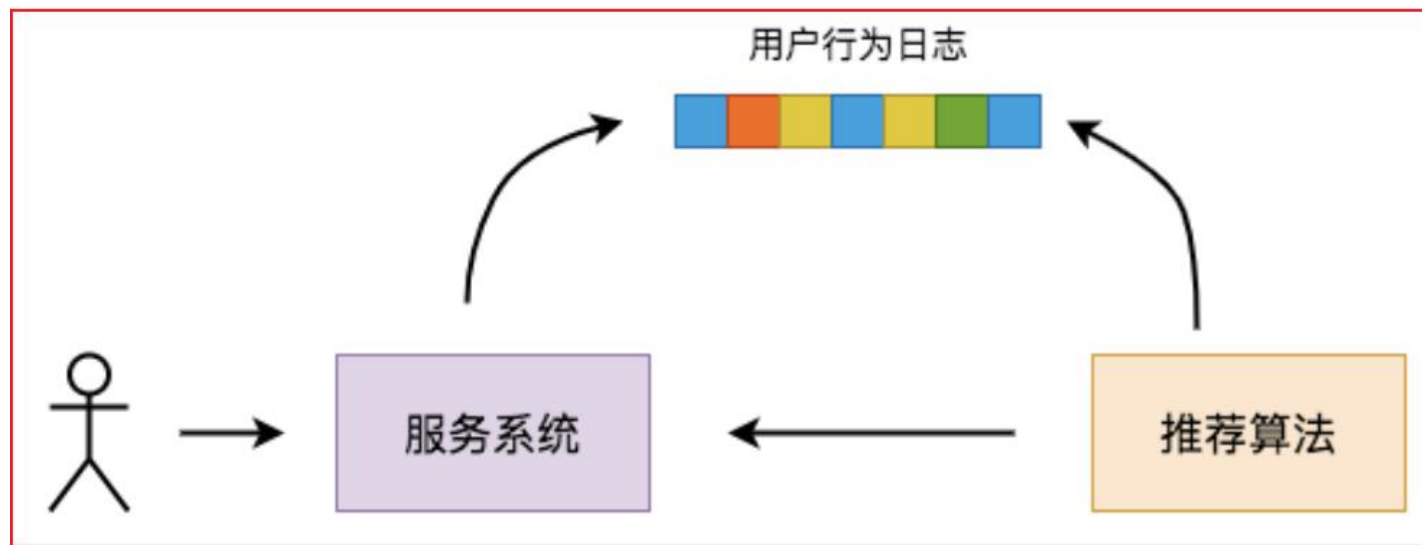
02

Pulsar架构原理(Broker与Bookkeeper)

- 深入理解Pulsar分层存储
- 深入理解Bookkeeper架构
- Pulsar如何实现生产和消费流程
- Pulsar跨机房复制

深入理解Pulsar分层存储

在一些流数据用例场景中，用户希望将数据长时间存储在流中。虽然 Apache Pulsar 对 topic backlog 的大小没有限制，但将所有数据存储在 Pulsar 中较长时间，存储成本比较大。分层存储支持在不影响终端用户的条件下，将较旧的数据移动到长期存储中。



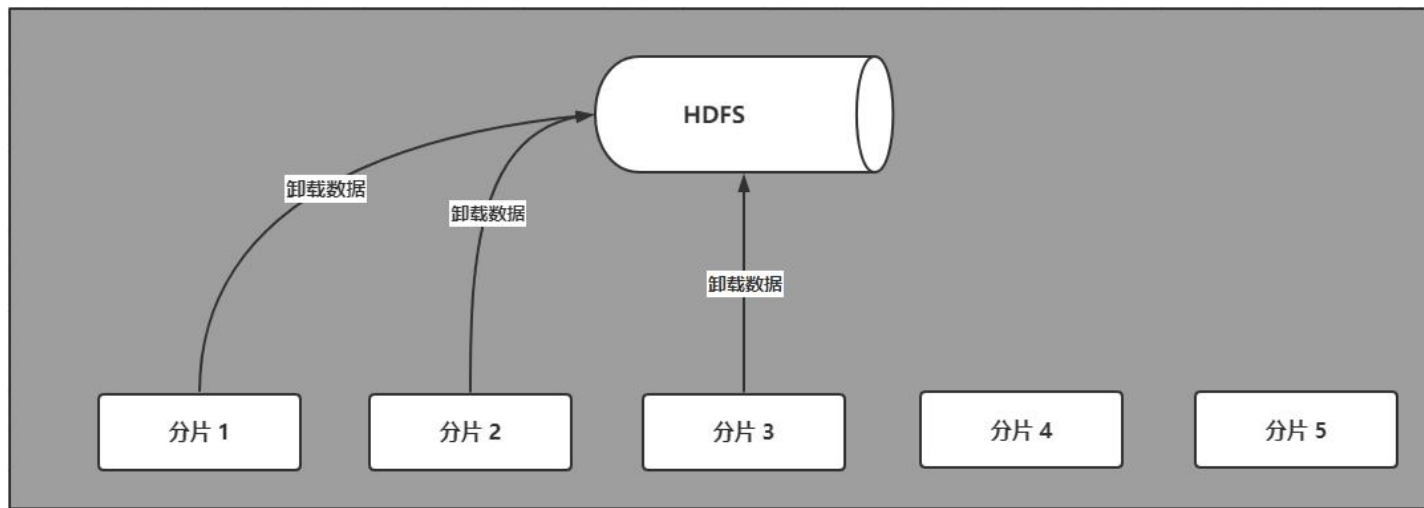
深入理解Pulsar分层存储

Pulsar 允许用户存储任意大小的 topic backlog。当集群将要耗尽空间时，用户只需添加新的存储节点，系统将会自动重新平衡数据。但是，这样的操作运行一段时间后，运维成本十分昂贵。

Pulsar 通过提供分层存储（Apache Pulsar 2.1 起新增的特性）减少了成本/大小的损失。分层存储为用户提供大小不受限制的 backlog，且无需添加存储节点；卸载较旧的 topic 数据到长期存储中，长期存储的成本比在 Pulsar 集群中存储的成本低一个数量级。对于终端用户来说，消费存储在 Pulsar 集群或分层存储中的 topic 数据没有明显差别。位于 Pulsar 集群和分层存储中的 topic 生产和消费消息的方式也完全相同。

Pulsar 通过分片架构实现了分层存储。Pulsar topic 的消息日志由一系列分片组成。序列中的最后一个分片是 Pulsar 当前写入的分片。当前序列之前的所有分片都已封装，也就是说，这些分片中的数据不可变。由于数据不可变，因此可以轻易地将数据复制到另一个存储系统，而不必担心一致性的问题。复制完成后，可以立即更新消息日志元数据中的数据指针，并且可以删除 Pulsar 在 Apache BookKeeper 中存储的数据副本。

深入理解Pulsar分层存储



Pulsar 目前支持通过 Amazon S3、GCS (Google Cloud Storage)、HDFS 进行长期存储。

整体配置操作也是非常简单的，只需要在broker.conf中配置卸载地址和路径，并开启卸载自动运行即可，详细配置大家可参考：<https://pulsar.apache.org/docs/en/cookbooks-tiered-storage/>



02

Pulsar架构原理(Broker与Bookkeeper)

- 深入理解Pulsar分层存储
- **深入理解Bookkeeper架构**
- Pulsar如何实现生产和消费流程
- Pulsar跨机房复制

深入理解Apache Bookkeeper架构

Apache BookKeeper 是企业级存储系统，旨在保证高持久性、一致性与低延迟。

自 2011 年起，BookKeeper 开始在 Apache ZooKeeper 下作为子项目孵化，并于 2015 年 1 月作为顶级项目成功问世。

企业级的实时存储平台需要具备的特点：

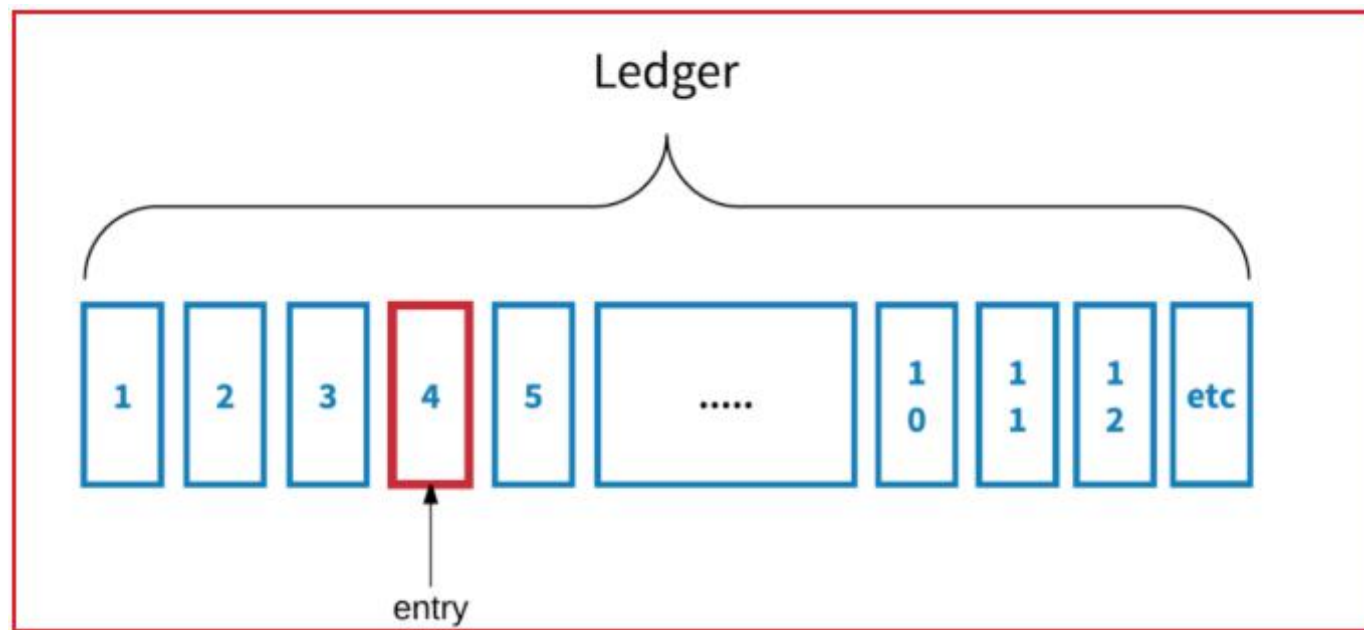
- 以极低的延迟（小于 5 毫秒）读写 entry 流
- 能够持久、一致、容错地存储数据
- 在写数据时，能够进行流式传输或追尾传输
- 有效地存储、访问历史数据与实时数据

BookKeeper 的设计完全符合以上要求，并广泛用于多种用例：分布式系统提供高可用性或多副本，在单个集群中或多个集群间（多个数据中心）提供跨机器复制，为发布/订阅（pub-sub）消息系统提供存储服务，为流工作存储不可变对象

Bookkeeper相关名词概念

BookKeeper 复制并持久存储日志流。日志流是形成良好序列的记录流。

Bookkeeper中比较核心的就两个元素：日志(ledger/stream)和记录(entry)



Bookkeeper相关名词概念

记录(entry)

数据以不可分割记录的序列，而不是单个字节写入 Apache BookKeeper 的日志。记录是 BookKeeper 中最小的 I/O 单元，也被称作地址单元。单条记录中包含与该记录相关或分配给该记录的序列号（例如递增的长数）。

客户端总是从特定记录开始读取，或者追尾序列。也就是说，客户端通过监听序列来寻找下一条要添加到日志中的记录。客户端可以单次接收单条记录，也可以接收包含多条记录的数据块。序列号也可以用于随机检索记录。

日志(ledger/stream)

BookKeeper 中提供了两个表示日志存储的名词：一个是 **ledger**（又称日志段）；另一个是 **stream**（又称日志流）。

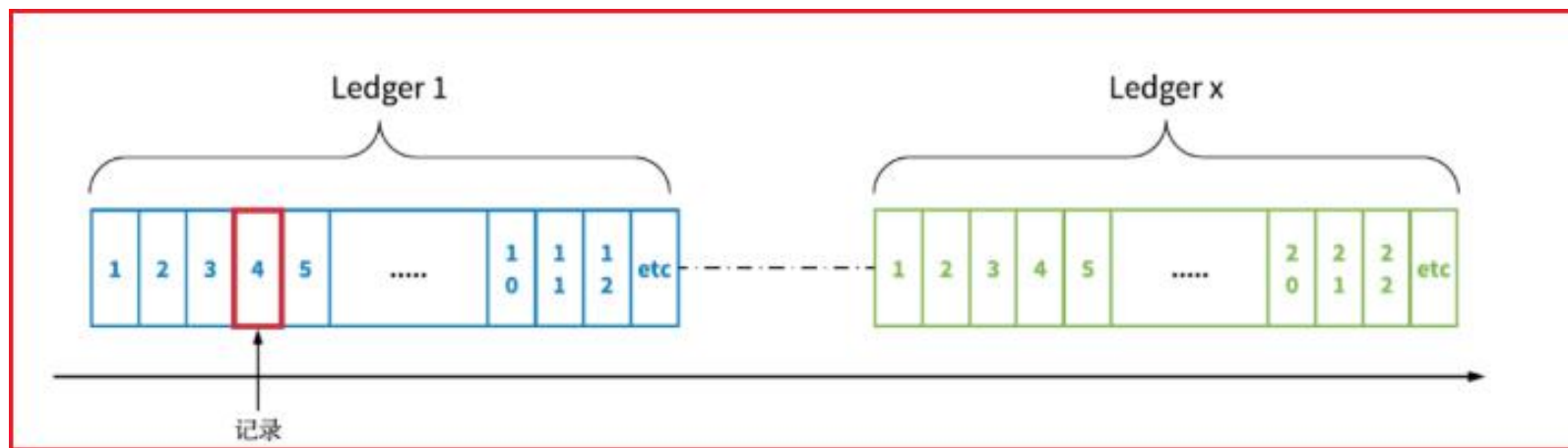
Ledger 用于记录或存储一系列数据记录（日志）。当客户端主动关闭或者当充当 writer 的客户端宕机时，正在写入此 ledger 的记录会丢失，而之前存储在 ledger 中的数据不会丢失。Ledger 一旦被关闭就不可变，也就是说，不允许向已关闭的 ledger 中添加数据记录（日志）。

Bookkeeper相关名词概念

Stream（又称日志流）是无界、无限的数据记录序列。默认情况下，stream 永远不会丢失。stream 和 ledger 有所不同。在追加记录时，ledger 只能运行一次，而 stream 可以运行多次。

一个 stream 由多个 ledger 组成；每个 ledger 根据基于时间或空间的滚动策略循环。在 stream 被删除之前，stream 有可能存在相对较长的时间（几天、几个月，甚至几年）。Stream 的主要数据保留机制是截断，包括根据基于时间或空间的保留策略删除最早的 ledger。

Ledger 和 stream 为历史数据和实时数据提供统一的存储抽象。在写入数据时，日志流流式传输或追尾传输实时数据记录。存储在 ledger 的实时数据成为历史数据。累积在 stream 中的数据不受单机容量的限制。



Bookkeeper相关名词概念

命名空间

通常情况下，用户在命名空间分类、管理日志流。命名空间是租户用来创建 stream 的一种机制，也是一个部署或管理单元。用户可以配置命名空间级别的数据放置策略。

同一命名空间的所有 stream 都拥有相同的命名空间的设置，并将记录存放在根据数据放置策略配置的存储节点中。这为同时管理多个 stream 的机制提供了强有力的支持。

Bookies:

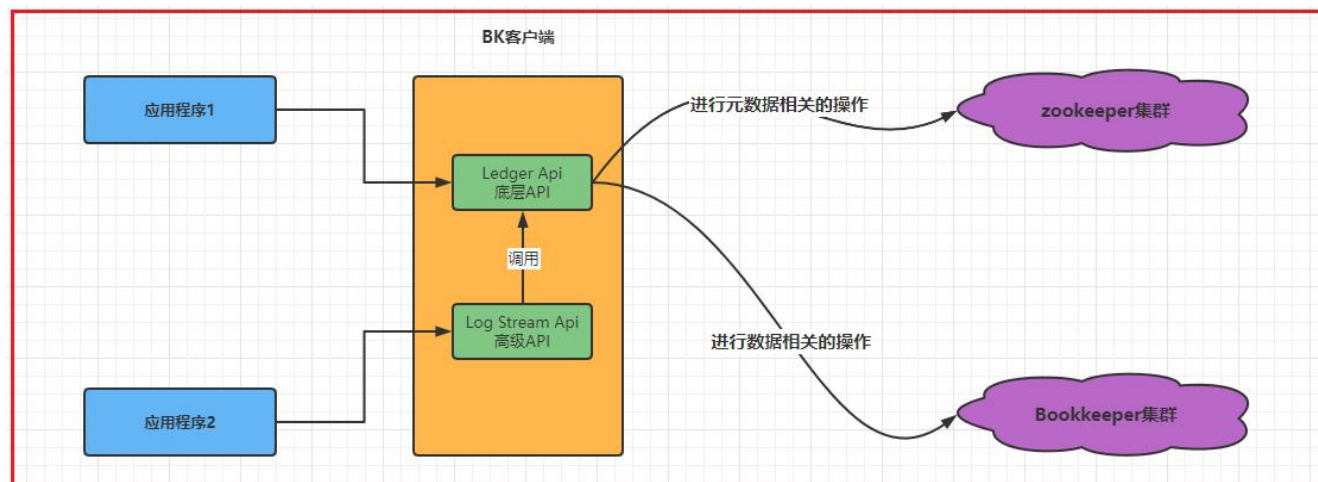
Bookies 即存储服务器。一个 bookie 是一个单独的 BookKeeper 存储服务器，用于存储数据记录。BookKeeper 跨 bookies 复制并存储数据 entries。出于性能考虑，单个 bookie 上存储 ledger 段，而不是整个 ledger。

因此，bookie 就像是整个集成的一部分。对于任意给定 ledger L，集成指存储 L 中 entries 的一组 bookies。将 entries 写入 ledger 时，entries 就会跨集成分段（写入 bookies 的一个分组而不是所有的 bookies）。

Bookkeeper相关名词概念

元数据

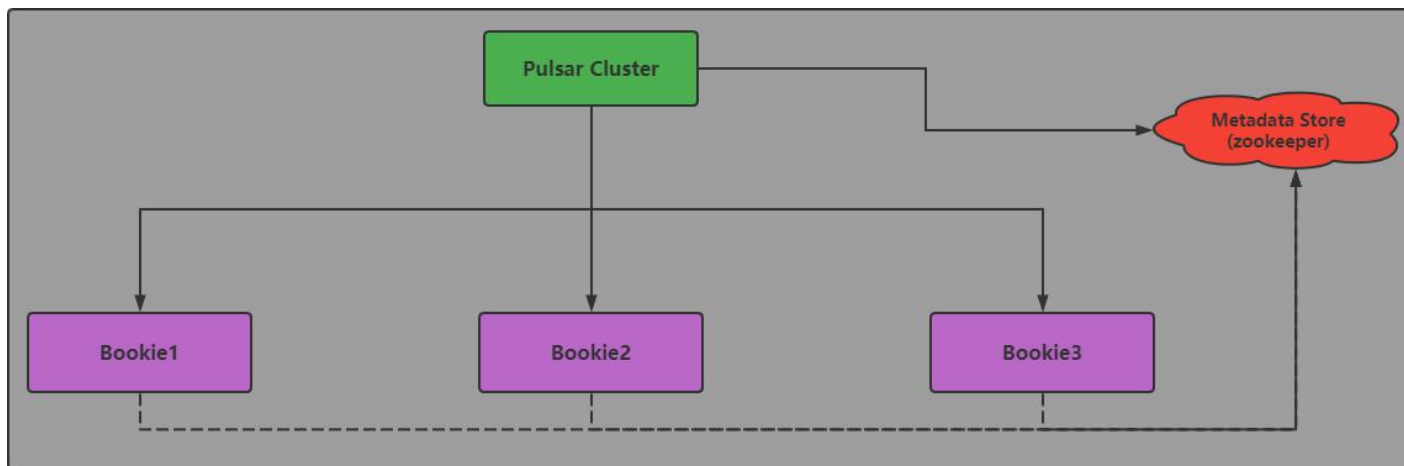
BookKeeper 需要元数据存储服务，用来存储 ledger 与可用 bookie 的相关信息。目前，BookKeeper 利用 ZooKeeper 来完成这项工作（除了数据存储服务外，还包括一些协调、配置管理任务等）。



1. 典型的 BookKeeper 安装包括元数据存储区（如 ZooKeeper）、bookie 集群，以及通过提供的客户端库与 bookie 交互的多个客户端。
2. 为便于客户端的识别，bookie 会将自己广播到元数据存储区。
3. Bookie 会与元数据存储区交互，作为回收站收集已删除数据。
4. 应用程序通过提供的客户端库与 BookKeeper 交互（使用 ledger API 或 DistributedLog Stream API）
5. 应用程序 1 需要对 ledger 进行粒度控制，以便直接使用 ledger API。
6. 应用程序 2 不需要较低级别 ledger 控制，因此使用更加简化的日志流 API。

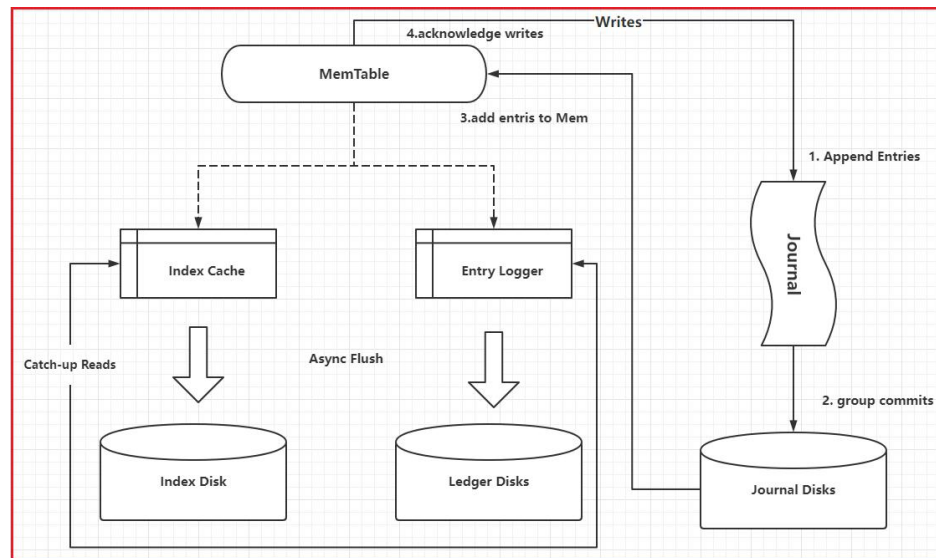
Bookkeeper整体架构

Bookkeeper的元数据存储 ----metadata store，目前是由zookeeper进行，用于存储leader ID对应的元数据信息



而集群中的 bookie 用来存储这些 ledger 对应的 entry，所有的 bookie 会注册到 BookKeeper 上，由客户端去发现并采取相应的操作。BookKeeper 的客户端主要是实现一些与一致性、策略性相关的逻辑。

Bookkeeper整体架构



- Bookie 的实现，依靠 journal(类似于WAL预写日志) 和 ledger storage, Bookie 利用 journal 进行所有写的操作。在追加多条 entry（来自不同的 ledger）的过程中，journal 都在发挥着它的持久化作用。这样做的优点是无论 ledger 来自何处，Bookie 只负责按顺序将entry写到journal文件里，不会进行随机访问。
- 当一个 journal 文件写满后，Bookie 会自动开启一个新的 journal 文件，继续按顺序填补 entry 。
- 但问题是，用户无法在 journal 里查询某条 entry。所以如果应用到读请求时，就需要「索引」功能来达到更高效的过程。
- 为了让各组件独立完成任务，没有在 journal 上建立索引功能，而是在 bookie 端维持了一个「write cache」，在内存里进行一个写缓存。在 journal 里运行结束后，会放置到 write cache 里。
- 经过 write cache 过程后，Bookie对 entry 进行重新排序，按 ledger 的来源划分整理entry，以便确保在缓存变满的过程中，entry 可以按照 ledger 的顺序排队。
- 当缓存变满后，bookie 会把整个 write cache 冲到磁盘里。Flush 的过程又重新整理了几个目录，用来保留相关的映射关系。一个是 entry log，用来存储 value。同时维护另一个 ledger index，用来记录 entry id 的位置。
- 默认的 ledger storage 有两类：DB ledger storage 和 Sorted ledger storage。本质上，这两类ledger storage的实现途径是一样的，只是在处理索引存储时不太一样。

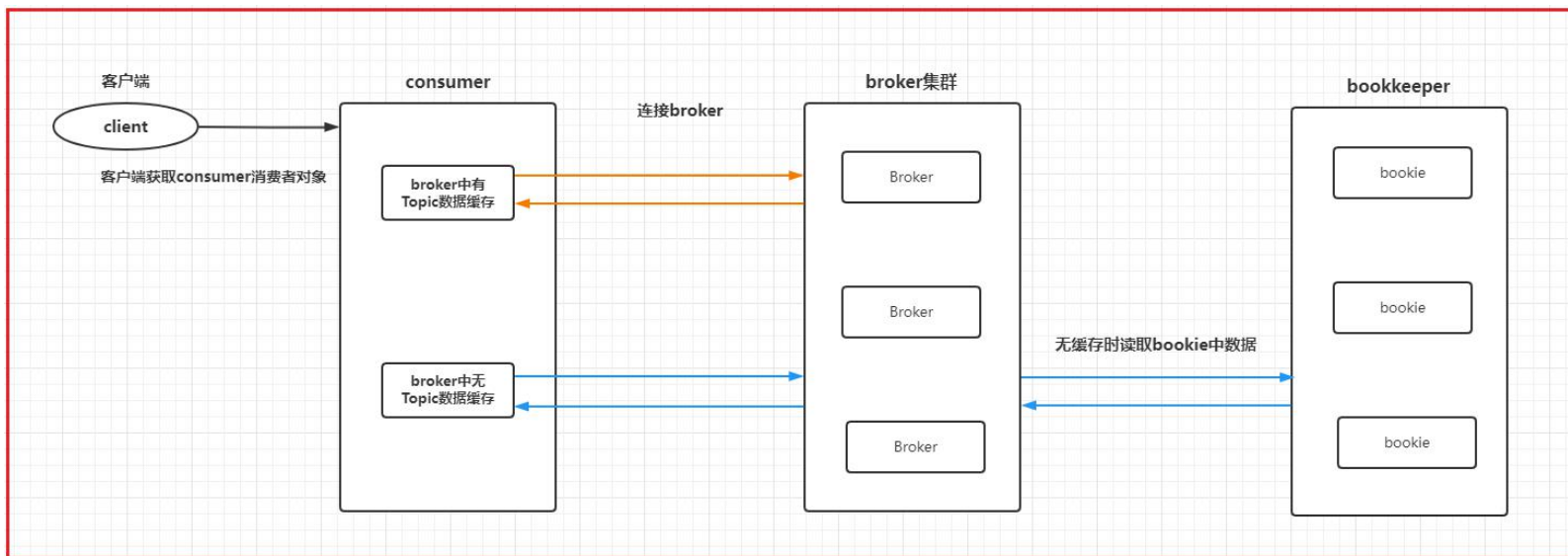


02

Pulsar架构原理(Broker与Bookkeeper)

- 深入理解Pulsar分层存储
- 深入理解Bookkeeper架构
- Pulsar如何实现生产和消费流程
- Pulsar如何实现Exactly-once(精准一次)
- Pulsar跨机房复制

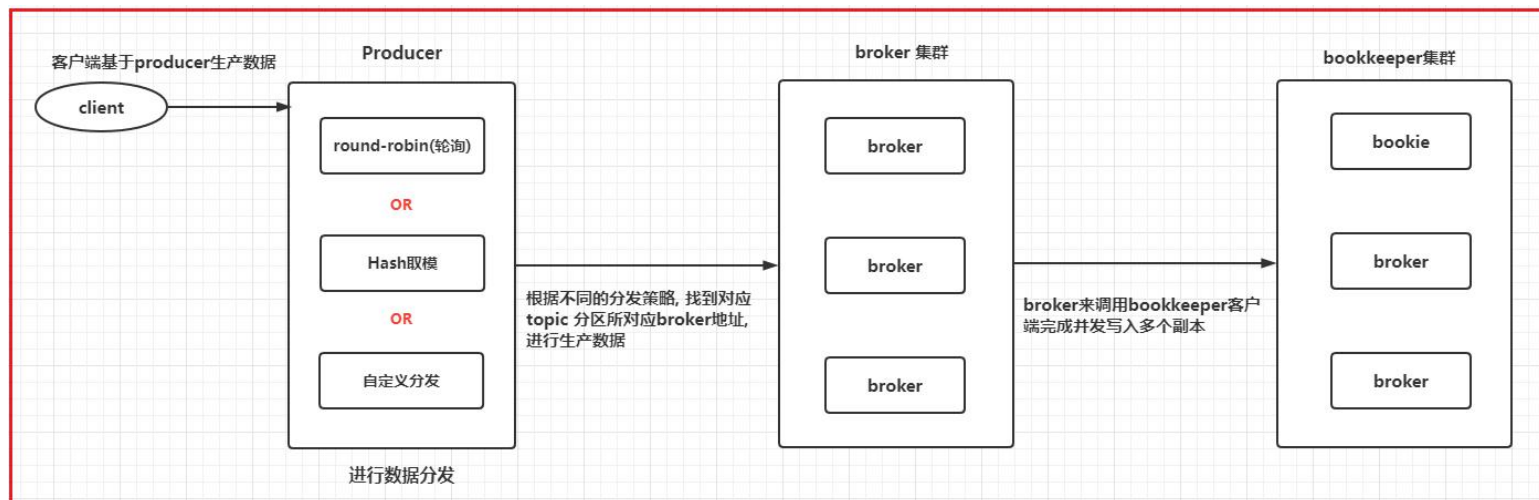
Pulsar数据生产流程



Consumer在消费数据时候，主要有二种情况，一种为broker中已经缓存了消息，一种为broker中没有缓存信息

- 1) 消费者连接broker地址，根据要读取的对应topic的分配，确定要连接的最终的broker地址，如果没有指定分片，那么就连接每一个分片对应的broker地址
- 2) 对应的broker首先判断消息是否已经有缓存数据，如果存在，直接从内存中采用推的方式发送给消费者，将消息放置在一个 receiver 队列中，消费端从队列中读取即可，如果没有缓存，此时broker端通过bookkeeper的客户端到bookie中读取数据(内部可以读取任意副本的数据)

Pulsar数据读取流程



- 1) 客户端调用pulsar提供给客户端的API，进行数据的生产操作，将生产的消息传递给producer
- 2) 在生产端内部有一个MessageWriter的类，基于这个类实现数据分发操作，默认方案为round-robin(轮询), 同时为了提高效率，在一定的时间内，只会选择一个partition
除了支持轮询方案外，如果在传递消息指定key，会采用hash取模的方式确定要发送到那个partition，同时pulsar支持自定义分发策略
- 3) 客户端在此连接broker，根据要发送的partition获取对应服务的broker节点
- 4) broker收到消息后调用bookkeeper的客户端并发去写多个副本
- 5) broker端会等待bookkeeper写入完成，当broker收到所有副本的ack之后，会认为这条消息已经写入成功，broker会返回客户端，告知这条消息已经被持久化完成

说明：整个写入操作，客户端不会跟zookeeper打交道, 也不会和bookkeeper打交道，只需要和broker即可

Pulsar数据读写故障处理流程

- 生产端产生失败：

当出现 [发消息网络断开, broker宕机] 等情况时候，这个时候producer有 pending 队列，会在设置的超时时间内进行重试策略

- Broker端出现宕机

因为broker是没有状态的，所以它不保存任何数据，一旦宕机后，topic的管理权会被其他broker掌管，这个时候，服务会被快速恢复

- Bookkeeper出现宕机

存储节点只负责数据存储，bookkeeper本身是一个集群，故如果只挂掉一个bookie，并不影响，所以broker是不会感知的，除非所有的bookie都挂掉，没有足够的副本去写入数据。

- 消费端产生失败

一个订阅同时只有一个消费者，但是可以拥有多个备份消费者，一旦主消费者故障，则备份消费者接管，进行消费即可，同时pulsar还支持一个分区对应多个消费者，或者一个消费端对应多个分片的情况

同时只要消息没有被消费者所消费，在pulsar中消息就没有变成确认状态，下次依然是可以再次消费的



02

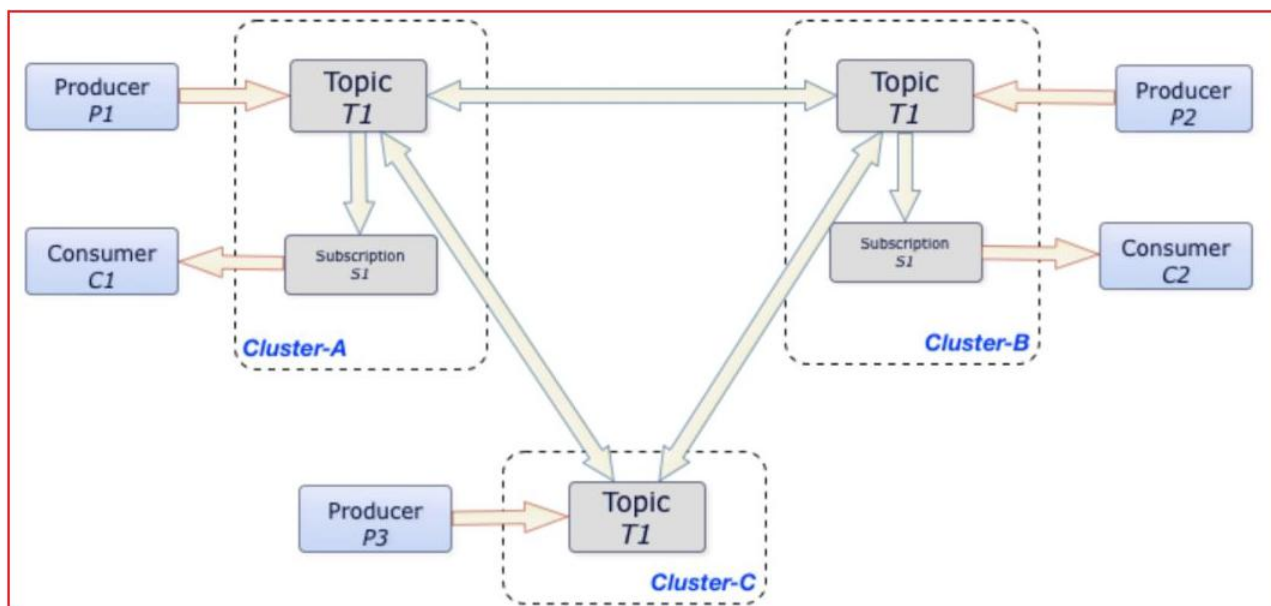
Pulsar架构原理(Broker与Bookkeeper)

- 深入理解Pulsar分层存储
- 深入理解Bookkeeper架构
- Pulsar如何实现生产和消费流程
- Pulsar跨机房复制

Pulsar跨机房复制

在大型的分布式系统中，都会涉及到跨多个数据中心的需求，通常会使用跨地域复制机制提供额外的冗余防止服务无法正常运作。Apache Pulsar 的跨地域多机房互备特性，是 Pulsar 企业级特性的重要组成部分，它在保证数据稳定可靠的同时，为用户提供了便捷的操作和管理。

Pulsar 自带的跨地域复制机制（Geo-Replication）可以提供一种全连接的异步复制。



Pulsar跨机房复制

在上图系统中，有三个数据中心：Cluster-A、Cluster-B、Cluster-C。用户创建的一个 Topic 主题 T1 设置了跨越三个数据中心做互备。在三个数据中心中，分别有三个生产者：P1、P2、P3，它们往主题 T1 中发布消息；有两个消费者：C1、C2，订阅了这个主题，接收主题中的消息。

当消息由本数据中心的生产者发布成功后，会立即复制到其他两个数据中心。消息复制完成后，消费者不仅可以收到本数据中心产生的消息，也可以收到从其他数据中心复制过来的消息。

它的工作机制是在 Broker 内部，为跨地域的数据复制启动了一组内嵌的额外生产者和消费者。当外部消息产生后，内嵌的消费者会读取消息；读取完成后，调用内嵌的生产者将消息立即发送到远端的数据中心。

跨地域复制需要设置“租户”在数据中心之间的访问权限。

在配置了跨地域复制后，每个发送进来的消息，首先被保存在本地集群中；然后异步地推送到远端的集群。如果本地集群和远端集群之间没有网络问题，消息会被立即推送给远端集群。这样端到端的发送延迟主要由集群之间网络的决定。

在图示中，无论生产者（Producer）P1、P2 和 P3 在什么时候分别将消息发布给 Cluster A、Cluster B 和 Cluster C 中的主题 T1，这些消息均会立刻复制到整个集群。一旦完成复制，消费者（Consumer）C1 和 C2 即可从自己所在的集群消耗这些消息，并且保持消息在每个 Producer 内部的发送顺序。

因为消息已经从其他远端集群发送到本地集群的 Topic，所以每个集群内部都会保留这个 Topic 中产生的所有消息。对于每个 Consumer 来说，Consumer 的订阅（subscription，维护 Consumer 对 Topic 的消费和 ack 的位置）是针对本地集群的 Topic，相当于 Consumer 消费本地集群的消息。

Pulsar跨机房复制 如何配置呢

说明：此处操作，是模拟以及构建了三个pulsar数据中心的方案(cluster1, cluster2, cluster3)

- 第一步: 首先创建一个租户, 并赋予三个数据中心的权限

```
$ bin/pulsar-admin tenants create my-tenant \  
--allowed-clusters cluster1, cluster2, cluster3
```

- 第二步: 创建namespace

```
$ bin/pulsar-admin namespaces create my-tenant/my-namespace
```

- 第三步: 设置namespace中topic在那些数据中心之间进行互备

```
$ bin/pulsar-admin namespaces set-clusters my-tenant/my-namespace \  
--clusters cluster1, cluster2, cluster3
```

说明: 在未来, 如果新增了数据中心, 或者关闭数据中心, 可以随时进行配置调整操作, 而且pulsar表示这样的操作并不会对流量有任何影响

以下模拟了一个新增数据中心的操作:

```
bin/pulsar-admin namespaces set-clusters my-tenant/my-namespace \  
--clusters cluster1, cluster2, cluster3
```



03

Pulsar Adaptors适配器

- Kafka 适配器
- Spark 适配器

Pulsar Adaptor on Kafka 适配器

Pulsar 为使用 Apache Kafka Java 客户端 API 编写的应用程序提供了一个简单的解决方案。

在生产者中，如果想不改变原有kafka的代码架构，就切换到Pulsar的平台中，那么Pulsar adaptor on kafka就变的非常的有用了，它可以帮助我们在不改变原有kafka的代码基础上，即可接入pulsar，但是需要注意，相关配置信息需要进行一些调整，例如：地址与topic

- 1- 需要导入Pulsar兼容kafka的依赖包

```
<dependency>
  <groupId>org.apache.pulsar</groupId>
  <artifactId>pulsar-client-kafka</artifactId>
  <version>2.8.1</version>
</dependency>

<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka-clients</artifactId>
  <version>2.4.1</version>
</dependency>
```

Pulsar Adaptor on Kafka 适配器

- 2- 编写生产者

```
public class KafkaAdaptorProducer {
    public static void main(String[] args) throws Exception {
        //1. 创建kafka生产者的核心类对象: KafkaProducer
        //1.1: 创建生产者配置对象: 设置相关配置
        Properties props = new Properties();
        props.put("bootstrap.servers", "pulsar://node1:6650,node2:6650,node3:6650");
        props.put("acks", "all"); // 消息的确认方案
        props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer"); // key序列化类型
        props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer"); // value 序列化类型
        Producer<String, String> producer = new KafkaProducer<>(props);
        //2. 发送数据
        for (int i = 0; i < 10; i++) {
            //2.1: 创建 生产者数据承载对象 一个对象代表是一条消息数据
            ProducerRecord<String, String> producerRecord = new
            ProducerRecord<>("persistent://public/default/txn_t1", Integer.toString(i), Integer.toString(i));
            producer.send(producerRecord).get();
        }
        //3. 释放资源
        producer.close();
    }
}
```


Pulsar Adaptor on Kafka 适配器

- 3- 编写消费者

```
public class KafkaAdaptorConsumer {
    public static void main(String[] args) {
        //1. 创建kafka的消费者的核心对象: KafkaConsumer
        //1.1: 创建消费者配置对象, 并设置相关的参数:
        Properties props = new Properties();
        props.setProperty("bootstrap.servers", "pulsar://node1:6650,node2:6650,node3:6650");
        props.setProperty("group.id", "test"); // 消费者组的 id
        props.setProperty("enable.auto.commit", "true"); // 是否启动消费者自动提交消费偏移量
        props.setProperty("auto.commit.interval.ms", "1000"); // 每间隔多长时间提交一次偏移量: 单位 毫秒
        props.setProperty("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer"); // key 反序列化
        props.setProperty("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer"); // val 发序列化
        KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
        //2. 给消费者设置订阅topic:
        consumer.subscribe(Arrays.asList("persistent://public/default/txn_t1"));
        //3. 循环获取相关的消息数据
        while (true) {
            //3.1: 从kafka中获取消息数据: 参数表示等待超时时间
            // 注意: 如果没有获取到数据, 返回一个空集合对象, 如果数据集合中有多个 ConsumerRecord 对象
            ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
            // 3.2 遍历ConsumerRecords 获取每一个 ConsumerRecord 对象 : ConsumerRecord 消费者数据承载对象, 一个对象就是一条消息
            for (ConsumerRecord<String, String> record : records) {
                String message = record.value();
                System.out.println("消息数据为:" + message);
            }
        }
    }
}
```

Pulsar Adaptor on Kafka 适配器

- 4- 先运行消费者, 进行监听, 然后运行生产者, 观察消费者是否可以正常消费到数据



```
KafkaAdaptorConsumer  KafkaAdaptorProducer
SLF4J: Failed to load class org.slf4j.impl.StaticLoggerBinder .
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
消息数据为:0
消息数据为:1
消息数据为:2
消息数据为:3
消息数据为:4
消息数据为:5
消息数据为:6
消息数据为:7
消息数据为:8
消息数据为:9
```



03

Pulsar Adaptors适配器

- Kafka 适配器
- Spark 适配器

Pulsar Adaptor on Spark 适配器

Pulsar 的 Spark Streaming 接收器是一个自定义的接收器，它使用 Apache Spark Streaming 能够从 Pulsar 接收原始数据。

应用程序可以通过 Spark Streaming receiver 接收 Resilient Distributed Dataset (RDD) 格式的数据，并可以通过多种方式对其进行处理。

- 1- 导入相关的依赖包

```
<dependency>
  <groupId>org.apache.pulsar</groupId>
  <artifactId>pulsar-spark</artifactId>
  <version>2.8.0</version>
</dependency>
```


Pulsar Adaptor on Spark 适配器

- 2- 编写spark的流式代码

```
String serviceUrl = "pulsar://localhost:6650/";
String topic = "persistent://public/default/test_src";
String subs = "test_sub";

SparkConf sparkConf = new SparkConf().setMaster("local[*]").setAppName("Pulsar Spark Example");
JavaStreamingContext jsc = new JavaStreamingContext(sparkConf, Durations.seconds(60));
ConsumerConfigurationData<byte[]> pulsarConf = new ConsumerConfigurationData();
Set<String> set = new HashSet<>();
set.add(topic);
pulsarConf.setTopicNames(set);
pulsarConf.setSubscriptionName(subs);
SparkStreamingPulsarReceiver pulsarReceiver = new SparkStreamingPulsarReceiver(
    serviceUrl,
    pulsarConf,
    new AuthenticationDisabled());

JavaReceiverInputDStream<byte[]> lineDStream = jsc.receiverStream(pulsarReceiver);
```



03

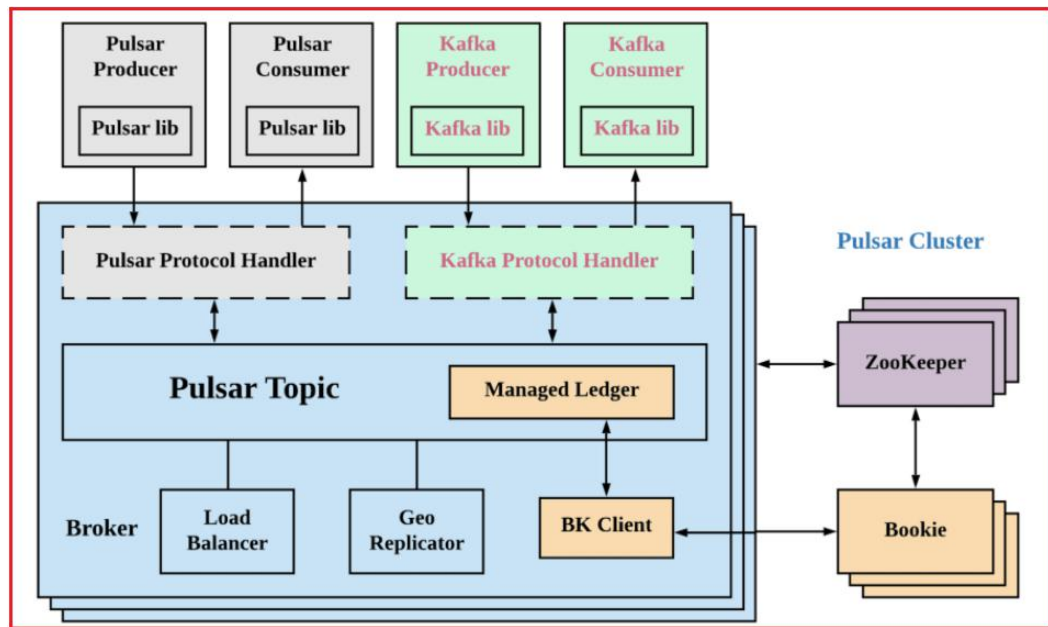
Apache Pulsar 可插拔协议

- kafka on Pulsar (KOP)
- RabbitMQ on Pulsar (AOP)

Apache Pulsar KOP

KoP (Kafka on Pulsar) 通过在 Pulsar 代理上引入 Kafka 协议处理程序，为 Apache Pulsar 带来了原生的 Apache Kafka 协议支持。通过将 KoP 协议处理程序添加到您现有的 Pulsar 集群，您可以将现有的 Kafka 应用程序和服务迁移到 Pulsar，而无需修改代码。这使 Kafka 应用程序能够利用 Pulsar 的强大功能。

KoP 作为 Pulsar 协议处理程序插件实现，协议名称为“kafka”，在 Pulsar broker 启动时加载。它通过在 Apache Pulsar 上提供原生 Kafka 协议支持，这样可以大大降低学习 Pulsar 的成本。基于 KOP 方案，整合两个流行的事件流生态系统软件。使用 Apache Pulsar 构建真正统一的事件流平台，以加速实时应用程序和服务的开发。



Apache Pulsar KOP -- 如何配置

- 1- 下载 KOP NAR包

<https://github.com/streamnative/kop/releases/download/v2.8.1.30/pulsar-protocol-handler-kafka-2.8.1.30.nar>

- 2- 将KOP NAR包上传到Pulsar的protocols目录中, 如果没有此目录, 直接创建即可

```
[root@node1 protocols]# ll
总用量 6692
-rw-r--r-- 1 root root 6851575 12月 21 16:12 pulsar-protocol-handler-kafka-2.8.1.30.nar
[root@node1 protocols]# pwd
/export/server/pulsar_2.8.1/protocols
```

- 3- 设置Kop的相关配置信息

```
cd /export/server/pulsar_2.8.1/conf
vim broker.conf
```

添加以下内容:

```
messagingProtocols=kafka
protocolHandlerDirectory=./protocols
kafkaListeners=PLAINTEXT://node1.itcast.cn:9092
brokerEntryMetadataInterceptors=org.apache.pulsar.common.intercept.AppendIndexMetadataInterceptor
```

修改以下内容: 147行 和 160行

```
allowAutoTopicCreationType=partitioned
brokerDeleteInactiveTopicsEnabled=false
```

Apache Pulsar KOP -- 如何配置

```
messagingProtocols=kafka
protocolHandlerDirectory=./protocols
kafkaListeners=PLAINTEXT://node1.itcast.cn:9092
brokerEntryMetadataInterceptors=org.apache.pulsar.common.intercept.AppendIndexMetadataInterceptor
# The type of topic that is allowed to be automatically created.(partitioned/non-partitioned)
allowAutoTopicCreationType=partitioned

# Enable subscription auto creation if new consumer connected (disable auto creation with value false)
allowAutoSubscriptionCreation=true

# The number of partitioned topics that is allowed to be automatically created if allowAutoTopicCreationType is set to partitioned.
defaultNumPartitions=1

# Enable the deletion of inactive topics
brokerDeleteInactiveTopicsEnabled=false
```

● 5- 说明

- 1- KOP的nar包各个broker节点都要上传操作
- 2- 配置文件每个Broker节点也是都要修改的, 其中 kafkaListeners各个节点要更改为自己的IP或者主机名

● 6- 重启各个Broker节点

Apache Pulsar KOP -- 如何使用

当Pulsar基于Kafka协议后,此时我们完全可以使用kafka的相关的命令的或者API来数据生产和消费,可以无痕迁移到Pulsar

- 1- 基于Kafka命令完成数据生产和消费

```
[root@node1 kafka_2.4.1]# ./bin/kafka-console-producer.sh --broker-list node1:9092 --topic t_kop01
>hello kop
>
```

构建生产者

进行生产消息

```
[root@node2 kafka_2.4.1]# ./bin/kafka-console-consumer.sh --bootstrap-server node1:9092,node2:9092,node3:9092 --topic t_kop01
hello kop
```

消费到生产者发送的数据

构建消费者

Apache Pulsar KOP -- 如何使用

- 2- 基于KAFKA的Java API方式_生产者

```
public class KafkaProducerTest {  
    public static void main(String[] args) {  
        //1. 创建kafka的生产者核心对象: KafkaProducer  
        //1.1: 为这个生产者设置相关的参数:  
        Properties props = new Properties();  
        props.put("bootstrap.servers", "node1:9092,node2:9092,node3:9092"); // 指定的kafka的地址列表  
        props.put("acks", "all"); // ack确认机制: 保证数据不丢失  
        props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer"); // key的序列化类  
        props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer"); // value 的序列化类  
  
        Producer<String, String> producer = new KafkaProducer<>(props);  
        //2. 进行数据的发送操作:  
        for (int i = 0; i < 10; i++) {  
            // 创建生产者的数据承载对象  
            ProducerRecord<String, String> producerRecord = new ProducerRecord<>("t-kop01", Integer.toString(i), Integer.toString(i));  
            producer.send(producerRecord);  
        }  
        //3. 释放资源  
        producer.close();  
    }  
}
```

Apache Pulsar KOP -- 如何使用

- 2- 基于KAFKA的Java API方式_生产者

```
public class KafkaConsumerTest {
    public static void main(String[] args) {
        //1. 创建消费端的核心对象: KafkaConsumer
        //1.1: 设置消费端相关的配置:
        Properties props = new Properties();
        props.setProperty("bootstrap.servers", "node1:9092,node2:9092,node3:9092"); // 设置kafka的服务地址
        props.setProperty("group.id", "test"); // 设置消费者组的id
        props.setProperty("enable.auto.commit", "true"); // 是否启动自动提交消费者的偏移量信息
        props.setProperty("auto.commit.interval.ms", "1000"); // 自动提交每次间隔的时间: 毫秒
        props.setProperty("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer"); // 反序列key
        props.setProperty("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer"); // 反序列value
        KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
        //2. 设置需要订阅的topic列表:
        consumer.subscribe(Arrays.asList("t-kop01"));
        //3. 循环监听, 从kafka中获取数据
        while (true) {
            ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(1000)); // 用于获取数据
            for (ConsumerRecord<String, String> record : records) {

                System.out.println("接收到消息: "+record.value());
            }
        }
    }
}
```


Apache Pulsar KOP -- 如何使用

```
E:\JAVA_soft\jdk1.8\bin\java ...
```

```
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
```

```
SLF4J: Defaulting to no-operation (NOP) logger implementation
```

```
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
```

```
数据发送完成
```

生产者生产数据

```
Process finished with exit code 0
```

kafkaConsumerTest

消费者收到消息

```
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
```

```
接收到消息: 0
```

```
接收到消息: 1
```

```
接收到消息: 2
```

```
接收到消息: 3
```

```
接收到消息: 4
```

```
接收到消息: 5
```

```
接收到消息: 6
```

```
接收到消息: 7
```

```
接收到消息: 8
```

```
接收到消息: 9
```



03

Apache Pulsar 可插拔协议

- kafka on Pulsar (KOP)
- AMQP协议 (RabbitMQ) on Pulsar (AOP)

Apache Pulsar AOP

AoP 是基于 Pulsar 特性实现的。但是，使用 Pulsar 和使用 AMQP 的方法是不同的。以下是 AoP 的一些限制。

- 目前，AoP 协议处理程序支持 AMQP0-9-1 协议，仅支持持久交换和持久队列。
- 一个 Vhost 由一个只能有一个包的命名空间支持。您需要提前为 Vhost 创建一个命名空间。
- Pulsar 2.6.1 或更高版本支持 AoP。

目前基于AMQP协议主要是 RabbitMQ为代表的消息队列,可以说 Pulsar的AOP方案主要是完成RabbitMQ到Pulsar之间的迁移工作，开发者可以基于AOP更方便的将原有在RabbitMQ部署业务迁移到Pulsar上

Apache Pulsar AOP 如何配置









- 1- 下载AOP的NAR包

<https://github.com/streamnative/aop/releases>

v2.8.1.30

Created by streamnative-ci

▼ Assets 8

 pulsar-protocol-handler-amqp-2.8.1.30.nar	156 MB
 pulsar-protocol-handler-amqp-2.8.1.30.nar.asc	833 Bytes
 pulsar-protocol-handler-amqp-2.8.1.30.nar.sha512	174 Bytes
 pulsar-protocol-handler-amqp-readme.md	12.4 KB
 pulsar-protocol-handler-amqp-readme.md.asc	833 Bytes
 pulsar-protocol-handler-amqp-readme.md.sha512	171 Bytes
 Source code (zip)	
 Source code (tar.gz)	

Apache Pulsar AOP 如何配置

- 2- 修改Pulsar的broker.conf配置文件

<https://github.com/streamnative/aop/releases>

```
[root@node1 protocols]# ll
总用量 166752
-rw-r--r-- 1 root root 163901078 12月 18 21:05 pulsar-protocol-handler-amqp-2.8.1.30.nar
-rw-r--r-- 1 root root 6851575 12月 21 16:12 pulsar-protocol-handler-kafka-2.8.1.30.nar
[root@node1 protocols]# pwd
/export/server/pulsar_2.8.1/protocols
```

- 3- 修改Pulsar的broker.conf配置文件

```
cd /export/server/pulsar_2.8.1/conf
vim broker.conf
```

```
messagingProtocols=amqp
protocolHandlerDirectory=./protocols
amqpListeners=amqp://node1.itcast.cn:5672
```

```
messagingProtocols=kafka,amqp 新增amqp协议, 如果没有集成kop, 直接新增此配置即可, 删除前序kafka
protocolHandlerDirectory=./protocols 指定协议nar包所在位置
kafkaListeners=PLAINTEXT://node1.itcast.cn:9092
brokerEntryMetadataInterceptors=org.apache.pulsar.common.intercept.AppendIndexMetadataInterceptor
amqpListeners=amqp://node1.itcast.cn:5672 新增amqp协议监听地址, 集群时写各个节点ip或者主机名, 如果local模式写127.0.0.1
```

Apache Pulsar AOP 如何配置

- 5- 说明

1- AOP的nar包各个broker节点都要上传操作

2- 配置文件每个Broker节点也是都要修改的, 其中 kafkaListeners各个节点要更改为自己的IP或者主机名

- 6- 重启各个Broker节点

Apache Pulsar AOP 如何使用

- 1- 在Pulsar中创建namespace

```
cd /export/server/pulsar_2.8.1/bin  
./pulsar-admin namespaces create public/vhost1
```

- 2- 构建项目. 加入maven相关的依赖

```
<dependency>  
  <groupId>com.rabbitmq</groupId>  
  <artifactId>amqp-client</artifactId>  
  <version>5.8.0</version>  
</dependency>
```

- 3- 编写代码,进行测试

```
// 1- 创建连接,构建传输管道  
ConnectionFactory connectionFactory = new ConnectionFactory();  
connectionFactory.setVirtualHost("vhost1");  
connectionFactory.setHost("node1");  
connectionFactory.setPort(5672);  
Connection connection = connectionFactory.newConnection();  
Channel channel = connection.createChannel();
```

Apache Pulsar AOP 如何使用

```
String exchange = "ex";
String queue = "qu";
// 设置 exchange
channel.exchangeDeclare(exchange, BuiltInExchangeType.FANOUT, true, false, false, null);
// 设置 queue
channel.queueDeclare(queue, true, false, false, null);
// 绑定 queue 和 exchange
channel.queueBind(queue, exchange, "");
// 向exchange生产数据
for (int i = 0; i < 100; i++) {
    channel.basicPublish(exchange, "", null, ("hello - " + i).getBytes());
}
// 从队列中获取相关的数据
CountDownLatch countDownLatch = new CountDownLatch(100);
channel.basicConsume(queue, true, new DefaultConsumer(channel) {
    @Override
    public void handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties, byte[] body) throws IOException {
        System.out.println("receive msg: " + new String(body));
        countDownLatch.countDown();
    }
});
countDownLatch.await();
// release resource
channel.close();
connection.close();
```


Apache Pulsar AOP 如何使用

```
Aop_RabbitMQ
E:\JAVA_soft\jdk1.8\bin\java ...
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
receive msg: hello - 1
receive msg: hello - 2
receive msg: hello - 3
receive msg: hello - 4
receive msg: hello - 5
receive msg: hello - 6
receive msg: hello - 7
```

正常消费到数据



传智教育旗下高端IT教育品牌