

一. Apache Flink Connector 简介

0. 什么是 Apache Flink?

Apache Flink 是由 Apache 软件基金会开发的开源流处理框架，其核心是用 Java 和 Scala 编写的分布式流数据流引擎。Flink 以数据并行和流水线方式执行任意流数据程序，Flink 的流水线运行时系统可以执行批处理和流处理程序。此外，Flink 的运行时本身也支持迭代算法的执行。

Apache Flink 的数据流编程模型在有限和无限数据集上提供单次事件（event-at-a-time）处理。在基础层面，Flink 程序由流和转换组成。

1. 什么是 Apache Flink Connector?

Flink 作为一个流批统一的计算引擎，需要从不同的第三方存储引擎中通过 Source 读取数据信息进行处理并通过 sink 写回到后续的存储引擎中。Connector 就作为 Flink 与数据库/数据流交互的连接件。

先看一看数据输入输出 Flink 的方式。

A、预定义的 source 和 sink:

基于文件的 source/sink: `readTextFile(path) / writeAsTest`

基于 socket 的 source/sink: `socketTextStream/ writeToSocket`

基于内存里的 collections（集合）、Iterator（迭代器): `fromCollection\fromElements / Print\ printToError`

B、Bundled Connectors(Flink 提供的)

Apache Kafka(source/sink)

Apache Cassandra(sink)

这两个方式也包括在我要分析的源码中，后续会详细介绍我们可以通过直接调用 API 使用。

C、Apache Bahir 中的连接器

D、Async I/O

这一次我负责的模块是 Connector-base, kafka, cassandra, hive 这四个部分。

2. 什么是 Flink Kafka Connector?

Kafka 是由 Apache 软件基金会开发的一个开源流处理平台，由 Scala 和 Java 编写。Kafka 是一种高吞吐量的分布式发布订阅消息系统，它可以处理消费者在网站中的所有动作流数据。

在生产环境中与 kafka 进行数据交互。用 kafka consumer 读取 kafka 生产环境数据，处理后用 kafka producer 写入 kafka 的 producer 分区。

3. 什么是 Flink Cassandra?

Cassandra 是一套开源分布式 NoSQL 数据库系统。它最初由 Facebook 开发，用于储存收件箱等简单格式数据，是一种流行的分布式结构化数据存储方案。

Cassandra connector 特点：提前写入日志（对非确定性程序，必须启用提前写日志）/ 检查点与容错（至少一次传输请求）

4. 什么是 Flink Kafka hive?

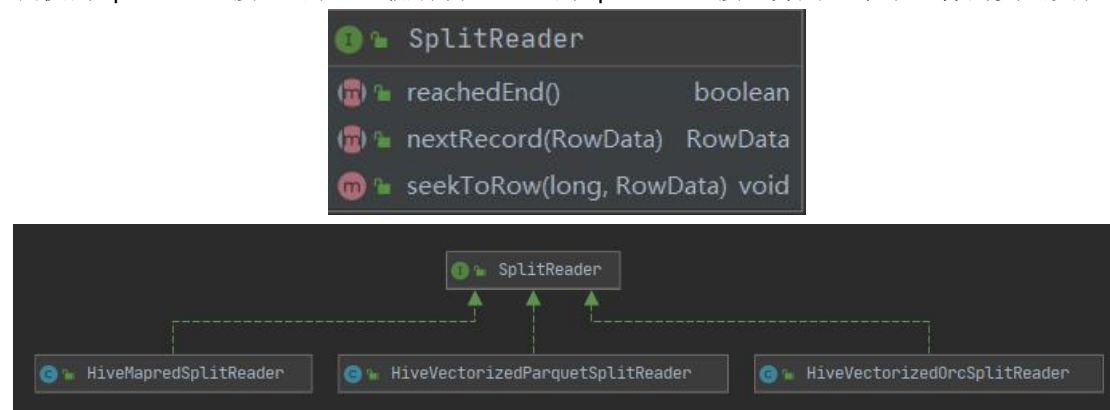
hive 是基于 Hadoop 的一个数据仓库工具, 用来进行数据提取、转化、加载, 这是一种可以存储、查询和分析存储在 Hadoop 中的大规模数据的机制。hive 数据仓库工具能将结构化的数据文件映射为一张数据库表, 并提供 SQL 查询功能, 能将 SQL 语句转变成 MapReduce 任务来执行。Hive 的优点是学习成本低, 可以通过类似 SQL 语句实现快速 MapReduce 统计, 使 MapReduce 变得更加简单, 而不必开发专门的 MapReduce 应用程序。hive 十分适合对数据仓库进行统计分析。

5. 相同与不同

开头提到, 既然我的任务是讲 base\kafka\Cassandra\hive 这四个 connector, 我就需要先表面分析一下它们四者的异同。

A、关于 splitter

Splitreader 是一个 base、hive、kafka 三者共有的接口, 其中, kafka 直接实现了 base 中提供的 splitter 接口, 而 hive 拥有自己独立的 splitter 接口并用三个不一样的类去实现。



其中, 类 `HiveMapredSplitReader` 通过 `hadoop mapreduce` 方式读取文件。

B、关于 sink、source 部分

Kafka 是一个处理平台, hive 是一个数据的统计分析工具, 两者都有 `sink/source` 两个独立类; 而 Cassandra 是一个数据库系统, 只有 `sink` 类

C、关于序列化器

三者都有独立的 `Deserializer`。

二. 主要功能分析与建模

0. Connector 主要类与接口简介

首先, 我们先介绍最基础的 `connector-base`。Connector-base 一共有 2 个主要模块, 分别是 `fetcher` 和 `reader`。其中使用到的类 `split Fetcher` 是 `connector` 的核心类。

`SplitFetcher` 类是一个内部的 `fetcher`, 它的作用是从外部系统轮询获取信息。它的两个参数都由泛型表示, 其中 `E` 代表读取元素类型 `element type`, `SplitT` 代表 `split` 的类型。之前提到 `flink` 是一个分布式系统, 其中最大的一个特点就是它的并行性, 故一个任务可以被分割成若干部分并行执行, 数据可以分成若干部分并行传输处理, 这就是 `split` 存在的原因。

```
1 public class SplitFetcher<E, SplitT extends SourceSplit> implements Runnable
```

还有一个要介绍的是接口 `SplitReader`, 它是一个用来读 `splits` 的接口, 实现它的类可以

读取 single split 或者 multiple splits，它的参数与 SplitFetcher 相同。

```
2 public interface SplitReader<E, SplitT extends SourceSplit>
```

1. 主要类属性与方法

根据 SplitFetcher 的类图我们可以看到，splitReader 接口是 splitfetcher 的其中一个属性。

splitFetcher 还有很多方法，其中 run 和 runonce 方法都用于运行 fetch 这一操作任务；Wakeup 方法用于唤醒执行 fetcher 任务的线程；shutdown 方法用于关掉 fetcher 任务。

SplitFetcher	
SplitFetcher(int, FutureCompletingBlockingQueue<RecordsWithSplitIds<E>>, SplitReader<E, SplitT>, Consumer<Throwable>, Runnable)	
run()	void
runOnce()	void
addSplits(List<SplitT>)	void
enqueueTask(SplitFetcherTask)	void
shutdown()	void
assignedSplits()	Map<String, SplitT>
isIdle()	boolean
shouldRunFetchTask()	boolean
wakeUp(boolean)	void
maybeEnqueueTask(SplitFetcherTask)	void
isRunningTask(SplitFetcherTask)	boolean
checkAndSetIdle()	void
shouldIdle()	boolean
splitReader	SplitReader<E, SplitT>

2. 需求建模

下面对我猜测的 splitfetcher 读取过程进行分析，判断如果 fetcher 需要从外部系统读取信息，应当如何建模：

82	需求建模
83	
84	[用例名称]
85	split读取器splitfetcher
86	[场景]
87	who:读取的splits、读取器reader、splits放置队列（比如queue）
88	where:存储器或者处理器
89	when:运行时

现在我们缺少的细节是如何启动 splitfetcher，如何将 split 分配给 splitfetcher，以及在 split 中的记录未被全部读取时，如何频繁地将 splitfetcher 唤醒这些细节。从而我们得到以下的用例描述：

93	需求建模
94	
95	[用例描述]
96	1.实例化splitfetcher和split(包括其中的records)
97	2.为splitfetcher分配一个执行fetch任务的线程
98	3.为splitfetcher分配一个唤醒它的线程
99	4.fetcher读取，wakeup线程通过某种计数方式唤醒fetcher直至split中的record读完
100	5.SplitFetcher关闭
101	6.线程回收
102	[用例价值]
103	完成split的读取
104	[约束和限制]
105	split数量和每个split中的records数量

找到其中的动词和名词：

```
108 需求建模
109
110 [名词]: splitfetcher、split、线程
111 [动词]: 执行、唤醒、读取、关闭、回收
```

从中抽象出 3 个类以及它们的属性和方法：

```
113 需求建模
114
115 [类]: splitfetcher
116 [方法]: run(执行)、fetch and read(读取)
117 [属性]: 状态
118
119 [类]: split
120 [方法]: 还没想到
121 [属性]: split类型、records类型、records数量
122
123 [类]: 线程
124 [方法]: join(回收)、wakeup(唤醒)
125 [属性]: 还没想到
```

3. 测试实例与流程

我们以 flink-connector-base/src/test 中的 SplitFetcherTest 类中的 testWakeup 方法讲解一下主要的流程。

A) 初始化属性和实例化类

这其中包括初始化 splits 的数量 numSplits，每个 splits 的记录数 numRecordsPerSplit，每读多少个记录数唤醒 fetcher 的记录数 WakeupRecordsInterval，元素队列 elementQueue，mock 对象 MockSplitReader 的实例以及由此实例化的 Fetcher。

```
final int numSplits = 3;
final int numRecordsPerSplit = 10_000;
final int wakeupRecordsInterval = 10;
final int numTotalRecords = numRecordsPerSplit * numSplits;

FutureCompletingBlockingQueue<RecordsWithSplitIds<int[]>> elementQueue =
    new FutureCompletingBlockingQueue<>(1);
SplitFetcher<int[], MockSourceSplit> fetcher =
    new SplitFetcher<>(
        0,
        elementQueue,
        new MockSplitReader(2, true),
        ExceptionUtils::rethrow,
        () -> {});
```

B) 初始化每个 split 的内容，包括了初始化 splits 以及其中的 records

```

List<MockSourceSplit> splits = new ArrayList<>();
for (int i = 0; i < numSplits; i++) {
    splits.add(new MockSourceSplit(i, 0, numRecordsPerSplit));
    int base = i * numRecordsPerSplit;
    for (int j = base; j < base + numRecordsPerSplit; j++) {
        splits.get(splits.size() - 1).addRecord(j);
    }
}

```

C) 调用 `splitfetcher` 的 `addSplits` 方法将 `splits` 添加到 `fetcher`，这一操作是异步的。

```
fetcher.addSplits(splits);
```

D) 为 `Fetcher` 分配一个执行任务的线程

```
Thread fetcherThread = new Thread(fetcher, "FetcherThread");
```

E) 为 `Fetcher` 分配一个时常唤醒它的进程。若 `Fetcher` 没有读取完所有的记录，则每完成一次规定的记录数通过调用 `fetcher` 的 `wakeup` 函数唤醒 `Fetcher` 一次（如果被阻塞则唤醒）。

```

AtomicInteger wakeupTimes = new AtomicInteger(0);
AtomicBoolean stop = new AtomicBoolean(false);
Thread wakeUpCaller = new Thread("Wakeup Caller") {
    @Override
    public void run() {
        int lastWakeup = 0;
        while (recordsRead.size() < numTotalRecords && !stop.get()) {
            int numRecordsRead = recordsRead.size();
            if (numRecordsRead >= lastWakeup + wakeupRecordsInterval) {
                fetcher.wakeup(false);
                wakeupTimes.incrementAndGet();
                lastWakeup = numRecordsRead;
            }
        }
    }
};

```

F) 调用 `fetcher` 的 `shutdown` 方法关闭它，并且等待回收所有线程。

```

stop.set(true);
fetcher.shutdown();
fetcherThread.join();

```

三、核心流程设计分析

0. 类间关系

下图给出了类间关系总览：

为了进一步地抽象化和模块化，`connectorObase` 实现了一些更为具体的接口和类。

`RecordswithsplitIds` 接口的作用有两个：辅助移动到下一个 `split`，和辅助读取同一个 `split` 的下一个 `records`。

`splitschange` 抽象类改变当前的 `split`，返回 `splits` 的列表。

`splitsreader` 与上述两者有依赖的关系。

`FutureCompletingBlockingQueue` 类是一个队列，它在 `Fetchtask` 类中作为一个元素队列

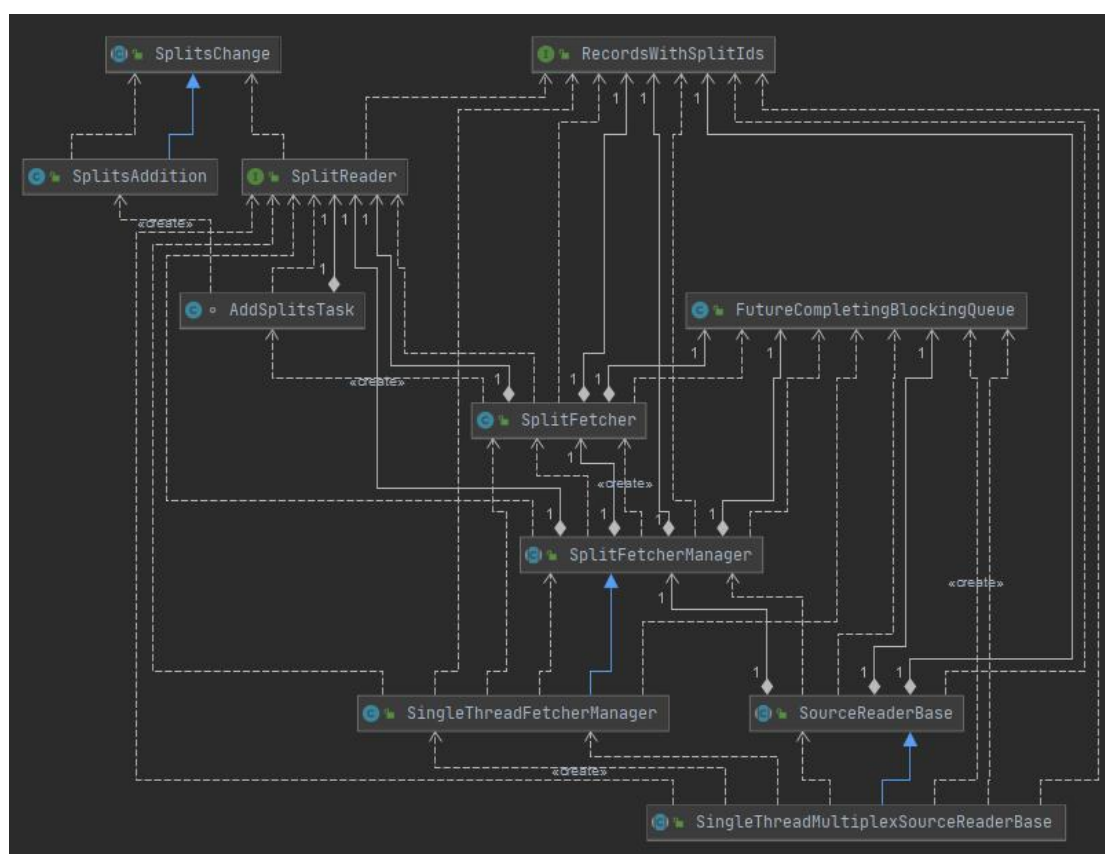
存在，存放取出的 records。

Fetchtask 类是一个读取记录到元素队列的类，而 Addsplitstask 是一个将任务分配到 splits 的类，两者的分工不同。

SplitFetcher 和上述两个 task 都有依赖关系，可以说是一个读取信息的完整任务。

SplitFetcherManager 和 SplitFetcher 是组合的关系，它负责启动以及控制 SplitFetcher 的生命周期，两者之间的关系十分密切。当然，SplitFetcherManager 有多种线程模型，而 SingleThreadFetcherManager 就是其中一类，它启用一个读取线程同时处理所有的 splits。

在 SingleThreadMultiplexSourceReaderBase 中，则分的更清晰，既可以 1 个线程代表一个 split,也可以 1 个线程同时处理所有 splits。



这些类与接口的关系充分体现了面向对象的重要思想，这就是抽象、继承、关联和多态。

首先我们看到这张图中的依赖线有很多两头标 1 的虚线，这代表了关联、聚合、组合这三种关系，其中组合关系最为重要，代表有关系的两者是非常相关，可以说是共存亡的关系。除了必要的对抽象类和继承类的继承之外，大部分类之间的关系都使用了依赖的关系，满足了组合/聚合复用原则，也就是尽量使用组合/聚合达到复用而非继承。

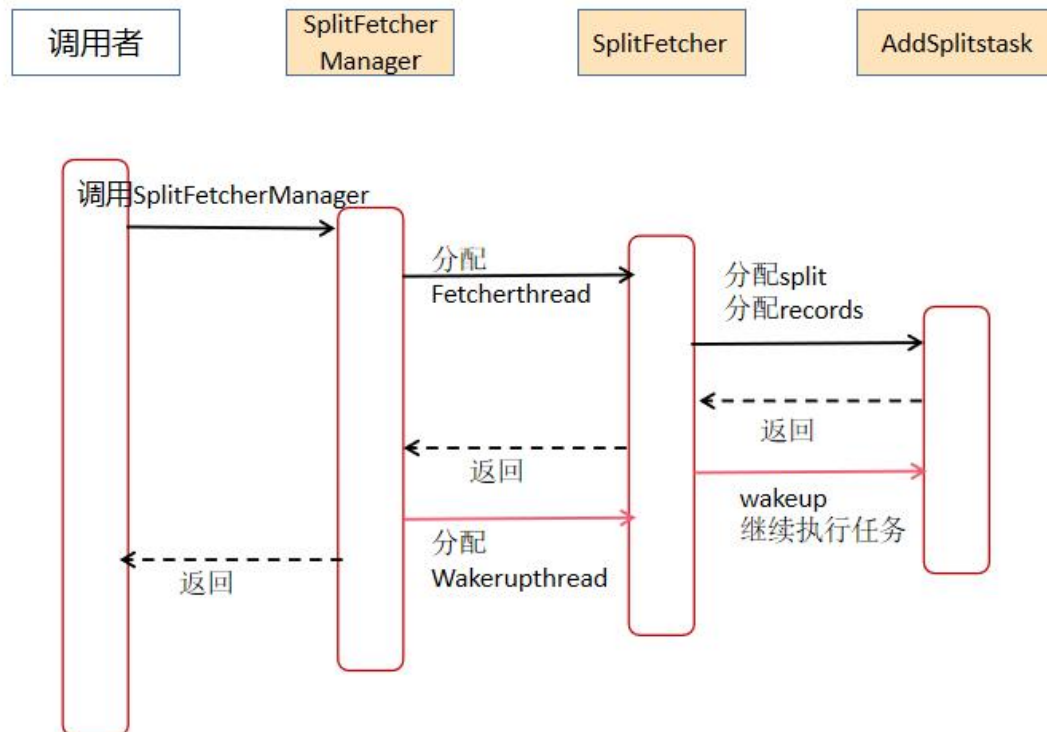
其次我们看到，SplitFetcher 依赖于 SplitReader 和 RecordsWithSplitIds 这两个接口，而不是用一个接口做所有的工作，满足了接口隔离原则，提供尽可能小的单独接口，而不是提供大的继承接口。

还可以看到，SplitCahnge 只做改变当前 Splits 这一件事情，而 FutureCompleteBlockingQueue 只做队列应当做的事，满足了单一职责原则，也就是每一个类

应当只完成一件事情。

1. 时序图

下面仅以 SplitFetcherManager、AddSplittask、SplitFetcher 类来绘制流程时序图

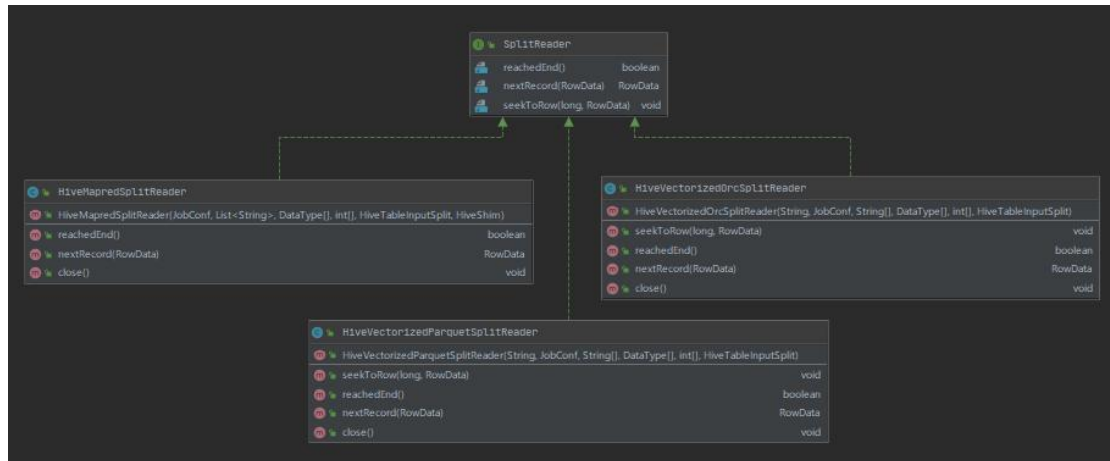


四、高级设计意图分析

0.Strategy 策略模型

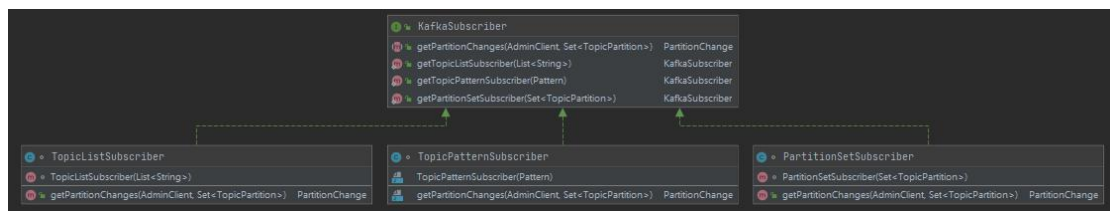
Strategy 模型的策略是，针对一组算法，将每一个算法封装到具有共同接口的独立的类中，从而使它们可以相互替换。为了避免在多个相似算法用 if-else 维护的情况，将这些算法封装成类，任意替换。

Hive-connector 的 splitreader 中运用了 Strategy 策略模型。可以看到 SplitReader 一共有三类方法，reachedEnd，nextRecord 和 seektoRow。而实现它的 3 个类，HiveMapredSplitReader、HiveVectorizedOrcSplitReader 以及 HiveVectorizedParquetSplitReader 都 override 了这三个方法。



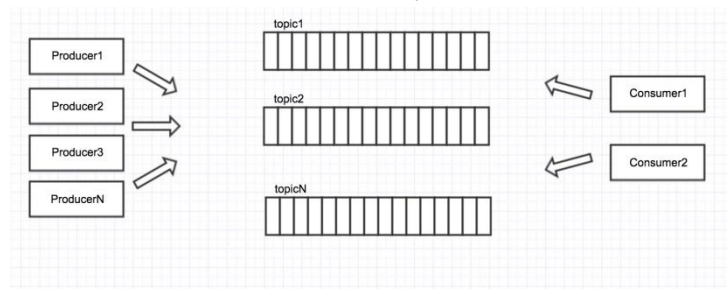
Kafka-connector 也体现了策略模型，kafka 消费者循序用不同的方式从主题进行消费。KafkaSubscriber 接口提供了自身这个统一的接口令 kafka source 能够支持多种订阅模式。

TopicListSubscriber、TopicPatternSubscriber、PartitionSetSubscriber 这三个类代表了三种不同的订阅策略，都 override 了 PartitionChange 方法。



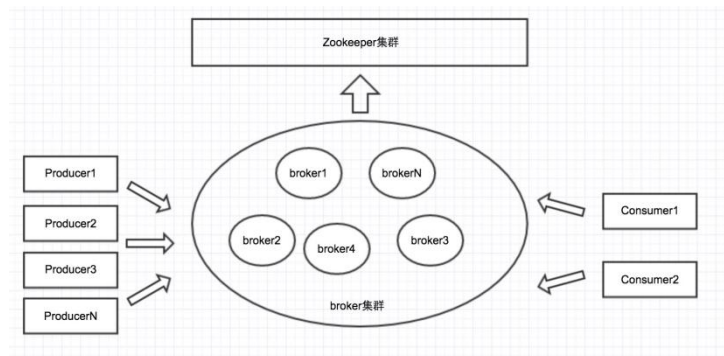
五. 番外篇（简单介绍一下我了解的 kafka）

Kafka 是高可靠的、持久化的消息队列。每个 topic 也就是自定义的一个队列，producer 往队列中放消息，consumer 从队列中取消息，topic 之间相互独立。

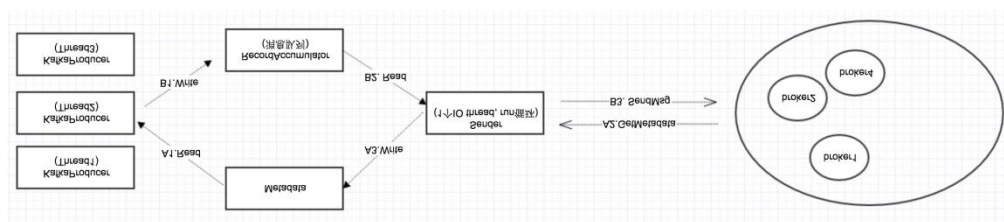


每个 broker 是一台物理机器，在上面运行 kafka server 的一个实例，所有这些 broker 实例组成 kafka 的服务器集群。

partition 是文件的目录。kafka 的 topic，在每个机器上，是用文件存储的。文件会分目录。为了使得 Kafka 的吞吐率可以水平扩展，物理上把 topic 分成一个或多个 partition，每个 partition 在物理上对应一个文件夹，该文件夹下存储。



数据流的走向是：**KafkaProducer** 把消息放到本地的消息队列 **RecordAccumulator**，然后一个后台线程 **Sender** 不断循环，把消息发给 **Kafka** 集群。**Sender** 从集群获取信息，然后更新 **Metadata**；**KafkaProducer** 先读取 **Metadata**，然后把消息放入队列。



六. 总结

感谢老师和助教这学期的精彩讲解，并热烈地迎接新年到来！！！！