# 16 | Apache Storm：最早的开源流计算框架

从今天开始，我们就正式步入第四个模块的学习了。在这个模块中，我们将根据前面课程中的各种流计算系统核心概念和关键技术点，来对多种开源流计算框架进行分析和验证。从而，实现一种从“形而上谓之道”向“形而下谓之器”的具象，让你在以后面对各种流计算工具和实际业务问题时，能够做到胸有成竹。

## 分析流计算框架的整体思路

在分析所有的流计算框架之前，让我们先制定一个整体的分析思路，这样能够让我们在后面的分析过程中，始终保持统一的思路，行事也会更有章法。

回顾前面课时的内容，我们从模块一中的接收实时流数据开始，然后在模块二时，构建起一个单节点的流计算应用，并实现了实时流计算中几种主要类型的数据处理，最后还将它扩展为一个分布式的系统。

我们实现的这个实时流计算系统，应该说具有了最基本的流计算框架雏形，它包含了流计算系统中最核心的几个要素：

- 首先是**流的本质**，也就是事件异步处理，并形成流水线；
- 然后是**流的描述**，也就是用DAG 拓扑结构描述流计算过程；
- 接着是**流量控制**，也就是反向压力，这是保证流计算系统稳定可靠运行的重要因素；
- 最后是**流的状态**，也就是流数据状态和流信息状态，这是流计算系统最关键的组件。

但是，如果是以一个成熟通用的流计算平台标准来看，我们开发的这个流计算框架，却还有很长的路要走。这是因为这个框架存在以下问题。

- 其一，它不是一个平台，只是一个编程框架。虽然可以部署为集群，但对于通常平台所要求的作业调度、资源管理等功能统统没有。
- 其二，流的描述不够抽象。这包含两层意思。一是只使用了比较底层的异步编程 API ，没提供更加上层的流计算编程 API ，比如 map、filter、reduce 等。二是 DAG的执行是在单一节点上完成，不像很多流计算平台 DAG的节点可以分配到不同计算节点上执行。
- 其三，作为流计算框架，只支持来一个事件就处理一个事件，不支持事件顺序校正，也不支持事件流按批次（各种窗口）处理。
- 其四，不支持比如至多一次（at most once）、至少一次（at least once）以及恰好一次（exactly once）等消息处理可靠性保证。
- 其五，缺乏消息处理失败时的应对策略。
- 另外，还有其他任何可能的问题……

或许，我们可以逐步地在现有框架上实现这些功能，但这将是一个漫长并具有很多不确定性的过程。因此，除了“闭门造车”外，我们还是需要研究已有的各种流计算解决方案。如无必要，而又能满足产品需求，还是应该尽量选择已有开源且相对成熟的流计算框架。

毕竟知己知彼，方能百战不殆。通过对这些开源流计算框架的学习，我们能够更加全面地理解流计算系统，把握流计算的发展状况和前进方向。

所以，我将从以下五个方面来考察各种流计算平台。

- 一是**系统架构**，因为理解一个流计算平台的设计架构，是使用这个流计算平台的基础。

- 二是**流的描述**，包括用于描述流计算过程的 DAG 和相关的 API 接口。
- 三是**流的处理**，包括与流的处理过程相关的 API，以及是否支持反向压力等。
- 四是**流的状态**，包括前面我们强调的流数据状态和流信息状态。
- 五是**消息处理可靠性**，包括流计算系统对消息传递的保证如何。

你看，上面五个方面中的二、三、四，不正是我们在前面三个模块中详细讨论过的内容吗？在本模块接下来的课时中，你会看到，上面这些概念和技术，正是各种开源流计算框架的核心组成部分！

所以接下来，就让我们开始一览众多开源流计算框架吧！不过今天，我们重点分析的是开源领域最早成名的流计算框架，也就是 Apache Storm。

## Apache Storm

Apache Storm（后续简称为 Storm）是一款由 Twitter 开源的大规模分布式流计算平台。 Storm 出现得比较早，算得上是分布式流计算平台的先行者。随着各种流计算平台后起之秀的崛起，以及它们对 Storm 带来的巨大冲击，Storm 自身也在不断尝试着改进和改变。不过，鉴于 Storm 可以说是最早被大家广泛接受的大规模分布式流计算框架，所以我们还是有必要先从Storm 开始分析。

## 系统架构

首先我们来看下 Storm 的系统架构。下图 1 是 Storm 的系统架构图。

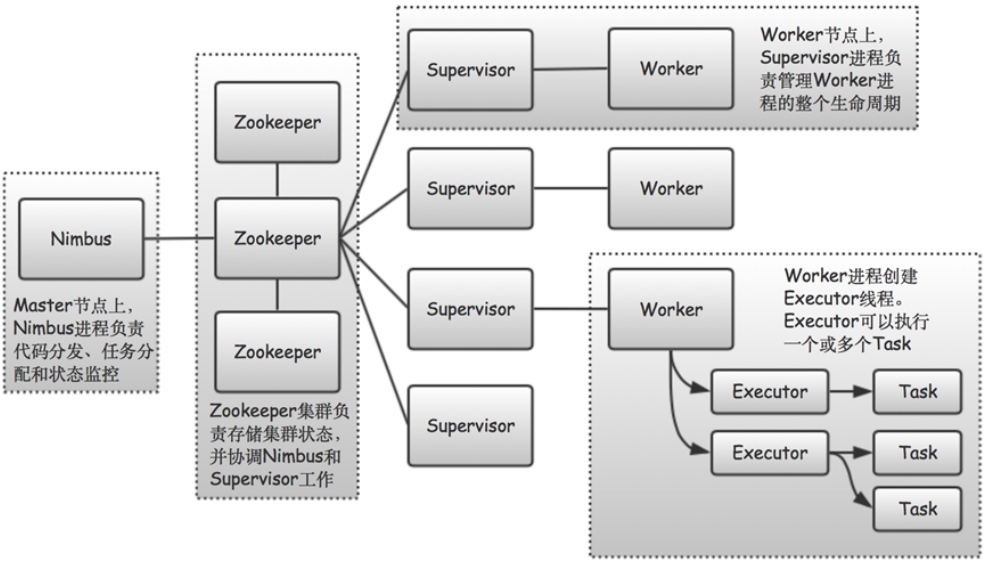



图 1 Storm系统架构图

从 Storm的系统架构图可以看出，Storm 集群由两种类型节点组成：Master 节点和 Worker 节点。

- Master 节点上运行 Nimbus 进程，用于代码分发、任务分配和状态监控。
- Worker 节点上运行 Supervisor 进程和Worker 进程，其中 Supervisor 进程负责管理 Worker 进程的整个生命周期，而 Worker 进程创建 Executor 线程用于执行具体任务（Task）。

在 Nimbus 和 Supervisor 之间，还需要通过 Zookeeper 来共享流计算作业状态，协调作业的调度和执行。

这里需要强调下，如果你是第一次接触这类大数据系统架构，请不要有任何担心！请勇敢地去理解它们。因为再复杂的系统，也无非就是由一个个的组件或进程构成，弄清楚每个组件或进程的作用后，就很容易理解整个系统了。

而且到最后，你会发现，不管是这些大数据架构，还是目前方兴未艾的微服务架构 Kubernetes 等，它们整体而言都是类似的，大差不差地最后都是“主从”架构，理解一个后就很容易理解其他的分布式系统架构了。

所以，如果你以前主要是做单体应用开发的话，也希望你能够从这里的 Storm 开始，逐步理解各种分布式系统架构。

## 流的描述

接下来，我们就来看看在 Storm 中是如何描述一个流计算过程的。下面的图 2 源自Storm 官网，它描述了 Storm Topology 的各个组件。我在原图基础上，对各个组件做了标注。

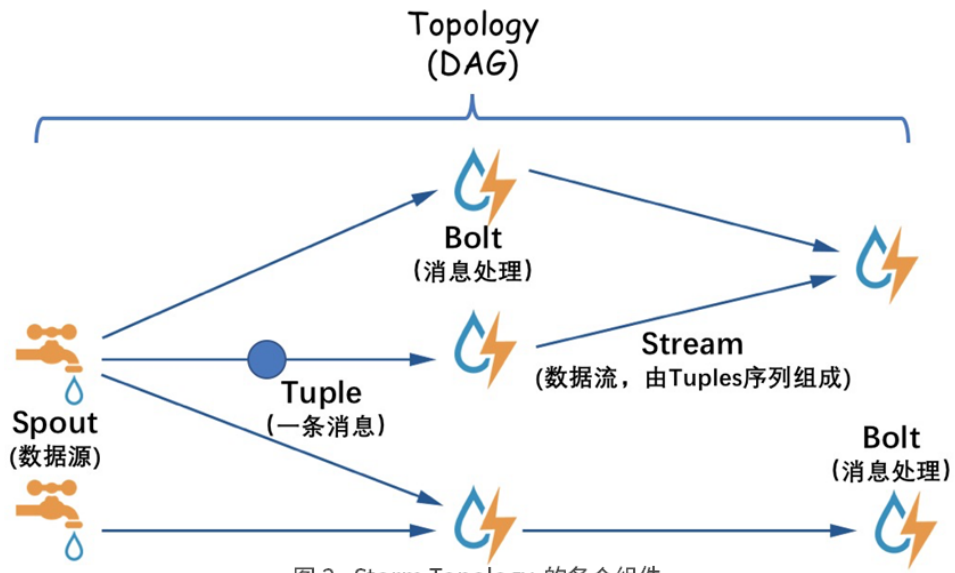



图 2　Storm Topology 的各个组件



在上面的图中，包含了 Storm 中 Topology、Tuple、Stream、Spout 和 Bolt这几个概念。

- 首先是**Topology**。Topology 就是我们在 05课时 中用来**描述流计算过程的DAG**，它完整地描述了流计算应用的执行过程。当 Topology 部署在 Storm 集群上并开始运行后，除非明确停止，否则它会一直运行下去。Topology 由 Spout、Bolt 和连接它们的 Stream 构成，其中Topology 的节点对应着 Spout 或 Bolt，而边则对应着 Stream。
- 然后是 **Tuple**。Tuple 是用于描述 Storm 中消息的，一个 Tuple就可以视为一条**消息**。
- 再然后是 **Stream**。Stream 是 Storm 中一个核心抽象概念，用于描述**消息流**。Stream 由 Tuple 构成，一个Stream 可以视为一组无边界的 Tuple 序列。
- 接着是 **Spout**。Spout 用于表示消息流的**输入源**。Spout 从外部数据源中读取数据，然后将其发送到消息流 Stream 中。
- 最后是**Bolt**。Bolt 是 Storm 进行消息处理的地方。Bolt 负责了消息的过滤、运算、聚类、关联、数据库访问等各种逻辑。开发者就是在 Bolt 中实现各种流处理逻辑。

从上面 Storm 的概念可以看出，与我们在模块二的内容相比，Topology 对应了 DAG，Stream 则相当于“队列”，而 Bolt 则相当于“线程”。你看，这是不是将一个抽象的流计算模式，落实为了具体的 Storm 实现?

## 流的处理

接下来我们再来看 Storm 中的流是怎么被处理的。

流的处理是指流计算应用中，输入的数据流，经过处理，最后输出到外部系统的过程。通常情况下，一个流计算应用会包含多个处理步骤，并且这些步骤的执行步调极有可能不一致。因此，还需要用反向压力来实现不同执行步骤间的流控。

早期版本的 Storm 使用 TopologyBuilder 来开始构建流计算应用。但是以新一代流计算框架的角度来看，基于 TopologyBuilder 的 API 在实际使用时并不直观和方便。

所以与时俱进的 Storm 从 2.0.0 版本开始，提供了更加现代化的流计算应用接口 Stream API。虽然目前 Stream API 仍然是实验阶段（用 @InterfaceStability.Unstable 注解标记着），但如果是从头开发一个新的 Storm 流计算应用的话，我建议还是直接使用 Stream API 就好了。因为 Stream API 风格的流计算编程接口，才是流计算应用开发的未来。

所以在接下来的讲解中，我就直接基于 Stream API，从流的输入、流的处理、流的输出和反向压力四个方面，来讨论 Storm 中流的执行过程。

首先是**流的输入**。Storm 是从 Spout 中输入数据流的，并且它提供了 StreamBuilder 来从 Spout 构建一个流。下面的代码（本课时完整代码请参考这里）就是一个典型的用 StreamBuilder 从 Spout构建 Stream 的例子。

```
public class DemoWordSpout extends BaseRichSpout {
    public void nextTuple() {
        Utils.sleep(100L);
        String[] words = new String[]{"apple", "orange", "banana", "mango", "pear"};
        Random rand = new Random();
        String word = words[rand.nextInt(words.length)];
        this._collector.emit(new Values(new Object[]{word}));
    }
}
StreamBuilder builder = new StreamBuilder();
Stream<String> words = builder.newStream(new DemoWordSpout(), new ValueMapper<String>(0));
```

在上面的代码中，Spout 的核心方法是 nextTuple，从名字上就可以看出这个方法是在逐条从消息源读取消息，并将消息表示为 Tuple。不同数据源的 nextTuple 方法实现方式并不相同。

另外需要说明的是，Spout 还有两个与消息传递可靠性和故障处理相关的方法，也就是 ack 和 fail。当消息发送成功时，可以调用 ack 从发送消息列表中删除已成功发送的消息。当消息发送失败时，则可以在 fail 中尝试重新发送或在最终失败时做出合适处理。

然后是**流的处理**。Storm 的 Stream API 与更新一代的流计算框架如 Spark Streaming、Flink等更加相似。总体而言，它提供了三类 API。

- 第一类是常用的流式处理操作，比如 filter、map、reduce、aggregate 等。
- 第二类是流数据状态相关的操作，比如 window、join、cogroup 等。
- 第三类则是流信息状态相关的操作，目前有 updateStateByKey 和 stateQuery。

下面是一个对 Stream 进行处理的例子。

```
wordCounts = words
    .mapToPair(w -> Pair.of(w, 1))
    .countByKey();
```

在上面的例子中，先用 mapToPair 将单词流 words 转化为计数元组流，然后再通过countByKey 转化为了单词计数流 wordCounts 。

接下来是**流的输出**。Storm的 Stream API 提供了将流输出到控制台、文件系统或数据库等外部系统的方法。目前提供的输出操作包括 print、peek、forEach 和 to 四个，其中：

- print 方法用于将流数据输出到标准输出流 stdout。
- peek 方法是对 stream的完全原样中继，并可以在中继时提供一段操作逻辑，因而 peek 方法可以用于方便地检测 stream 在任意阶段的状况。
- forEach 方法是最通用的输出方式，可以执行任意逻辑。
- to方法则允许将一个bolt 作为输出方法，可以方便地继承早期版本中已经存在的各种输出 bolt 实现。

下面是一个使用forEach 方法将流输出的例子。

```
wordCounts.forEach(new WordCountExample.Print2FileConsumer());
public static class Print2FileConsumer<T> implements Consumer<T> {
    public void appendToFile(Object line) {
        Files.write(Paths.get("/logs/console.log"),
                String.valueOf(line + "\n").getBytes(),
                StandardOpenOption.APPEND, StandardOpenOption.CREATE);
    }
    @Override
    public void accept(T input) {
        appendToFile(input);
    }
}
```

在上面的代码中，我们给 forEach 方法传入了一个 Print2FileConsumer 对象。在 Print2FileConsumer中，我们使用 appendToFile 方法，将文件数据写入了“/logs/console.log”文件。这样，就实现了将数据流输出到文件的目的。

最后是**反向压力**，Storm 支持反向压力。早期版本的 Storm 通过开启 acker 机制和 max.spout.pending 参数实现反向压力。当下游 Bolt 处理较慢，Spout 发送出但没有被确认的消息数超过 max.spout.pending 参数时，Spout 就暂停发送消息。

这种方式实现了反向压力，但却有些不算轻微的缺陷。一方面，静态配置 max.spout.pending 参数很难使得系统在运行时有最佳的反向压力性能。另一方面，这种反向压力实现，本质上只是在消息源头对消息发送速度做限制，而不是对流处理过程中各个阶段做反向压力，它会导致系统的处理速度，发生比较严重的抖动，降低系统的运行效率。

所以，在较新版本的 Storm中对上面的反向压力机制做了改进，除了监控 Spout发送出但没有被确认的消息数外，还监控每级 Bolt 接收队列的消息数量。当消息数超过阈值时，通过 Zookeeper 通知 Spout 暂停发送消息。这种方式实现了流处理过程中各个阶段反向压力的动态监控，能够更好地在运行时调整对 Spout的限速，降低了系统处理速度的抖动，也提高了系统的运行效率。

## 流的状态

接下来，我们再来看 Storm 中流的状态问题。前面在14 课时中，我们专门详细讨论过，流的状态分成两种：**流数据状态**和**流信息状态**。

我们先来看**流数据状态**。早期版本的 Storm 提供了 Trident、窗口（window）和自定义批处理三种有状态处理方案，我逐一讲解下。

**首先是Trident。** Trident是 Storm 中一套独特的应用构建机制，它将流数据切分成一个个的元组块（Tuple Batch），然后再将这些元组块，分发到集群中进行处理。

Trident 针对元组块的处理，提供了过滤、聚合、关联、分组、自定义函数等功能。特别是聚合、关联、分组等功能在实现过程中，**涉及状态保存的问题**。另外，Trident 在元组块处理过程中还可能失败，失败后需要重新处理，这个过程也会涉及**状态保存和事务一致性问题**。

因此 Trident 有针对性地**提供了一套状态接口（Trident StateAPI）来处理状态和事务一致性**。Trident 支持三种级别的 Spout 和 State：Transactional、Opaque Transactional 和 No-Transactional。其中 Transactional 提供了强一致性保证，Opaque Transactional 提供了弱一致性，No-Transactional 则无一致性保证。

**然后是窗口，Storm 支持 Bolt 按窗口处理数据，目前实现的窗口类型包括了滑动窗口**（Sliding Window）和**滚动窗口**（TumblingWindow）。

**最后是自定义批处理，** Storm 支持自定义批处理方式。Storm 系统内置了定时消息机制，通过每隔一段时间向 Bolt 发送嘀嗒元组（ticktuple），Bolt 在接收到嘀嗒元组后，可以根据需求自行决定什么时候处理数据、处理哪些数据等。在内置定时消息机制的基础上，我们可以实现各种自定义的批处理方式。比如，可以通过 tick 实现窗口功能（当然 Storm 本身已经支持），或实现类似于 Flink 中 watermark的功能（Storm 本身也已经支持）等。

不过，从 2.0.0 版本开始引入的Stream API 已经提供 window、join、cogroup 等**流数据状态相关的API**，这些 API 更加通用，使用起来也更方便，因此你可以直接使用这类 API 来开发 Storm 流计算应用。

说完流数据状态，我们再来看**流信息状态**。早期版本 Storm 中的Trident状态接口就包含了对流信息状态的支持，并且还支持了三种级别的事务一致性（前面讲流数据状态时已经提到过）。比如，使用 Trident 状态接口，就可以实现 word counts 的功能。

但是，由于 Trident状态与 Trident 支持的处理功能耦合太紧，这使得 Trident 状态接口的使用并不通用。比如**在非 Trident的Topology中就不能够使用 Trident 状态接口**了。所以在早期版本的 Storm 中，经常需要用户自行实现对流信息状态的管理。比如，使用 Redis 来记录事件发生的次数。

不过，最新版本 Storm的Stream API 已经开始引入更通用的**流信息状态接口**，目前提供的**updateStateByKey和stateQuery**就是这种尝试。这些流信息状态接口，对非 Trident的Topology 也能够使用，所以更加通用。

所以综上所述，不管是流数据状态，还是流信息状态，Storm 的 Stream API 都有更好更通用的支持，而且更符合现代化的流计算编程风格。

## 消息处理可靠性

最后，我们来看下 Storm 中消息处理可靠性的问题。

Storm 提供了不同级别的消息处理可靠性保证，包括尽力而为（best effort）、至少一次（at leastonce）和限于 Trident的精确一次（exactlyonce）。在实现消息处理可靠性机制时，Storm 使用了一种独特的技术，也就是**追踪消息是否被完全处理**。

在Storm 中，一条消息被完全处理，是指代表这条消息的元组以及由这个元组生成的子元组、孙元组以及各代重孙元组都被成功处理。反之，只要这些元组中有任何一个元组在指定时间内处理失败，那就认为这条消息是处理失败的。

因此，要使得 Storm 能够追踪消息是否被完全处理，需要在程序开发时，配合 Storm 系统做**两件额外的事情**。首先，在处理元组过程中生成了新元组时，需要**通过ack方法告知 Storm 系统**生成了该新元组。其次，当完成对一个元组的处理时，也需要**通过ack方法或 fail 方法告知 Storm系统**处理完了该元组。

鉴于以上过程还是比较烦琐的，所以我认为，在具体业务开发时，我们根据业务的实际需要，**选择合适的消息处理可靠性级别即可**。很多场景下并非一定要保证严格的exactlyonce，毕竟**越严格的消息处理可靠性级别，通常实现起来也会越复杂，性能损耗也会越大**。

## 小结

今天，我们从系统架构、流的描述、流的处理、流的状态和消息处理可靠性这五个方面，来对Storm 这个开源流计算框架进行了分析。我们可以看到，像 DAG、队列、线程、流数据状态、流信息状态、反向压力等，这些在前面的课时中详细讨论过的概念，在 Storm 里全部都找到了对应的实现，而且它们就是 Storm 的核心组成部分。

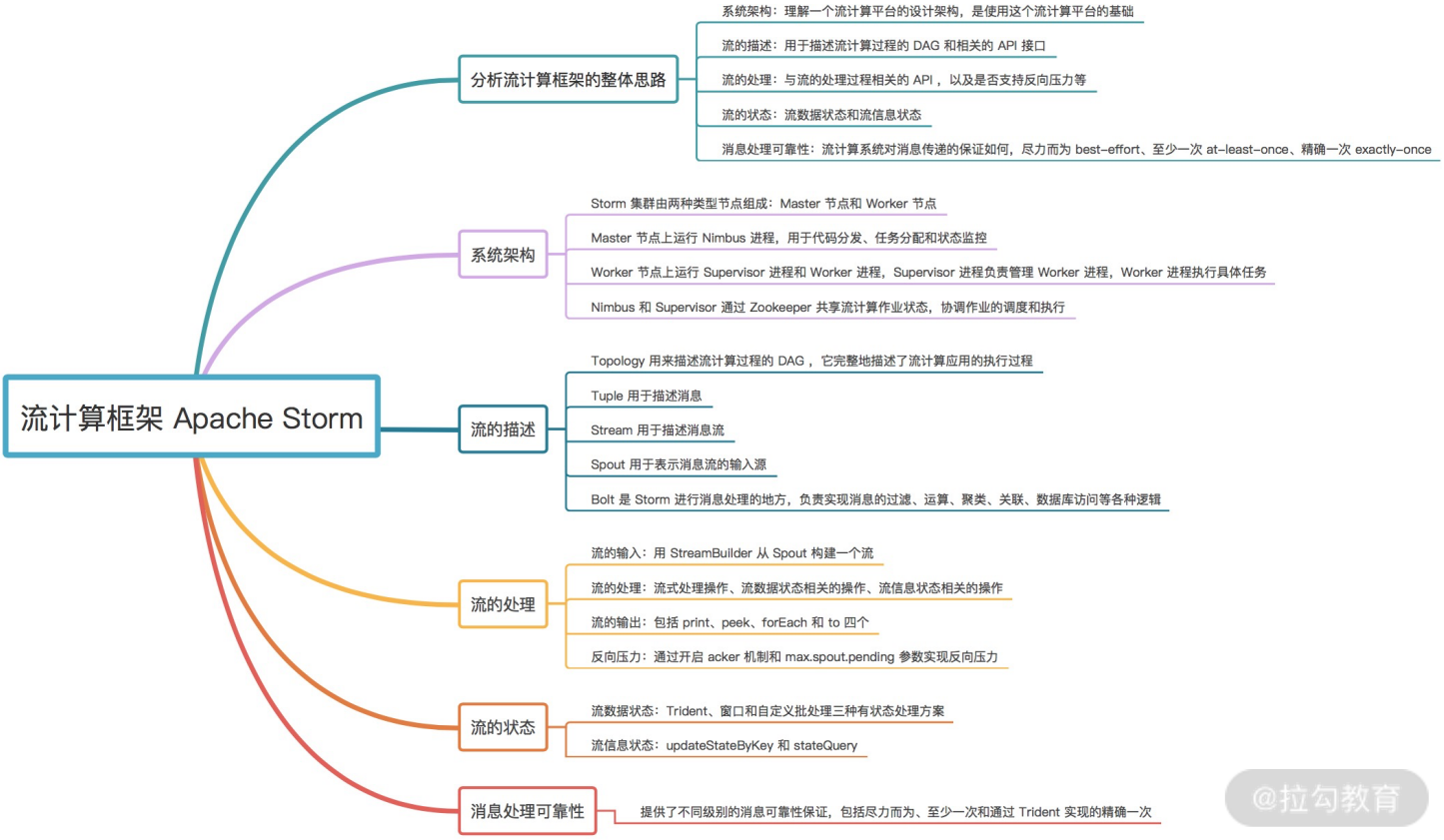
可以说，早期版本的 Storm，就像我们之前自己造的轮子一样，其编程接口其实是非常底层的，甚至显得很简陋。虽有 Trident，但是这种抽象将原来的“流”变成了“批”进行处理，与此对应，相关的API 也很不 stream-style。

好在，新版本的 Storm 已经在尝试支持更加现代化的 Stream API。这些 API的设计思路与其他现代化的开源流计算框架更加相似，理解起来也更加自然些。所以，如果是使用 Storm 开发一个新的流计算应用的话，你更应该直接使用 Stream API。

总的来说，Storm 是最早被大家广泛接受的框架，它也曾经风靡一时。但是，由于早期大家对流计算系统的认识还不够完善，Storm 在其设计和实现上还是有一些不足的，比如上层流式 API 和“状态”接口不够完善。虽然后面 Stream API的提出，以及对“状态”的支持，让 Storm 正在往正确的方向发展，但这并没有逆转其逐渐没落之势。甚至由于 Storm的继任者 Flink，实在是太过成功，以至于在方方面面都可以取代 Storm ，这就使得 Storm的处境更加尴尬。如果你是刚开始入门流计算领域的话，我推荐你从后面课时中会讲到的Flink 开始。

最后留一个小作业，你认为 Storm 的 Trident API 和 Stream API 有哪些相同之处和不同之处呢？它们各自有什么优缺点呢？可以将你的想法或问题写在留言区。

下面是本课时的脑图，以帮助你理解。


流计算框架 Apache Storm
分析流计算框架的整体思路
系统架构：理解一个流计算平台的设计架构，是使用这个流计算平台的基础
流的描述：用于描述流计算过程的 DAG 和相关的 API 接口
流的处理：与流的处理过程相关的 API，以及是否支持反向压力等
流的状态：流数据状态和流信息状态
消息处理可靠性：流计算系统对消息传递的保证如何，尽力而为 best-effort、至少一次 at-least-once、精确一次 exactly-once
系统架构
Storm 集群由两种类型节点组成：Master 节点和 Worker 节点
Master 节点上运行 Nimbus 进程，用于代码分发、任务分配和状态监控
Worker 节点上运行 Supervisor 进程和 Worker 进程，Supervisor 进程负责管理 Worker 进程，Worker 进程执行具体任务
Nimbus 和 Supervisor 通过 Zookeeper 共享流计算作业状态，协调作业的调度和执行
流的描述
Topology 用来描述流计算过程的 DAG，它完整地描述了流计算应用的执行过程
Tuple 用于描述消息
Stream 用于描述消息流
Spout 用于表示消息流的输入源
Bolt 是 Storm 进行消息处理的地方，负责实现消息的过滤、运算、聚类、关联、数据库访问等各种逻辑
流的处理
流的输入：用 StreamBuilder 从 Spout 构建一个流
流的处理：流式处理操作、流数据状态相关的操作、流信息状态相关的操作
流的输出：包括 print、peek、forEach 和 to 四个
反向压力：通过开启 acker 机制和 max.spout.pending 参数实现反向压力
流的状态
流数据状态：Trident、窗口和自定义批处理三种有状态处理方案
流信息状态：updateStateByKey 和 stateQuery
消息处理可靠性
提供了不同级别的消息可靠性保证，包括尽力而为、至少一次和通过 Trident 实现的精确一次