



Apache Flink 必知必会

零基础上手实战，7天搞定Flink



Apache Flink PMC及Committer
精心打造超强阵容实操教学





Flink 中文社区官方微信



Flink 社区技术交流钉钉群



Flink 中文社区 bilibili 官方账号



阿里云开发者“藏经阁”

海量电子书免费下载

目录

走进 Apache Flink	4
Stream Processing with Apache Flink	25
Flink Runtime Architecture	46
Fault-tolerance in Flink	64
Flink SQL _ Table 介绍与实战	94
PyFlink 快速上手	115
Flink Ecosystems	141
Flink Connector 详解	154

走进 Apache Flink

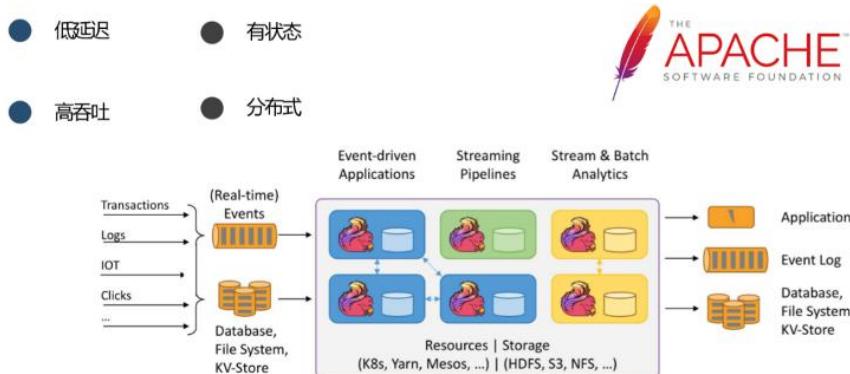
作者：李钰

Apache Flink PMC，阿里巴巴高级技术专家

本文由 Apache Flink PMC，阿里巴巴高级技术专家李钰分享，主要介绍 Apache Flink 的前世今生，内容大纲如下：

- 什么是 Apache Flink
- 为什么要学习 Apache Flink
- Apache Flink 典型应用场景
- Apache Flink 基本概念

一、什么是 Apache Flink



Apache Flink 是一个开源的基于流的有状态计算框架。它是分布式地执行的，具备低延迟、高吞吐的优秀性能，并且非常擅长处理有状态的复杂计算逻辑场景。

(一) Flink 的起源

Apache Flink 是 Apache 开源软件基金会的一个顶级项目，和许多 Apache 顶级项目一样，如 Spark 起源于 UC 伯克利的实验室，Flink 也是起源于非常有名的大学的实验室——柏林工业大学实验室。

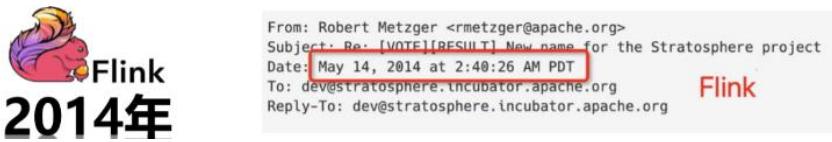


项目最初的名称为 Stratosphere，目标是要让大数据的处理看起来更加地简洁。项目初始的代码贡献者中，有很多至今仍活跃在 Apache 的项目管理委员会里，在社区里持续做出贡献。

```
jincheng:flink jincheng.suncs$ git remote -v
origin https://github.com/apache/flink.git (fetch)
origin https://github.com/apache/flink.git (push)
jincheng:flink jincheng.suncs$ git 3b88e30924
commit 3b88e30924268799c96317fe1bf95b9c6df6f80
Author: sruff <sruff@bart-2.local>
Date: Wed Dec 15 17:02:01 2010 +0100
test
jincheng:flink jincheng.suncs$ git merge origin/master
commit 3a524adba47955f
Merge: 86b95a0e643 e37el
Author: sruff <sruff@bart-2.local>
Date: Thu Dec 16 03:11 2010 +0100
Merge branch 'master' of https://www.stratosphere.eu/stratosphere
```

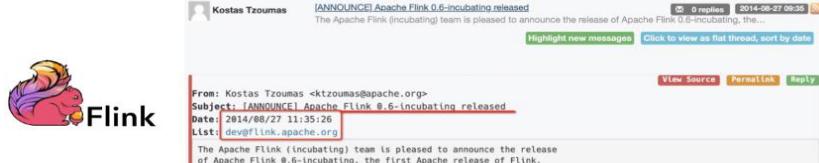


Stratosphere 项目于 2010 年发起，从它的 Git commit 日志里面可以看到，它
的第一行代码是在 2010 年的 12 月 15 日编写的。



2014 年 5 月，Stratosphere 项目被贡献到 Apache 软件基金会，作为孵化器
项目进行孵化，并更名为 Flink。

(二) Flink 的发展



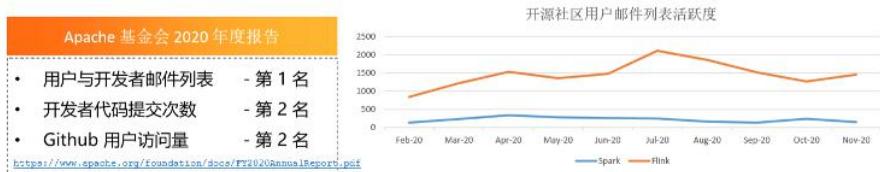
Flink 项目非常活跃，2014 年的 8 月 27 号发布了孵化器里的第一个版本 v0.6-
incubating。



由于 Flink 项目吸引了非常多贡献者参与，活跃度等方面也都非常优秀，它在 2014 年 12 月成为了 Apache 的顶级项目。

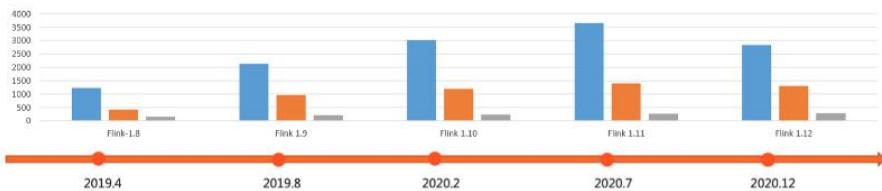
成为顶级项目之后，它在一个月之后发布了第一个 Release 版本 Flink 0.8.0。在此之后，Flink 基本保持 4 个月 1 个版本的节奏，发展到今天。

(三) Flink 的现状-Apache 社区最活跃的项目



发展至今，Flink 已成为 Apache 社区最活跃的大数据项目。它的用户与开发者邮件列表活跃度在 2020 年的 Apache 年度报告中排名第一。

如上方右图所示，与非常活跃的 Spark 项目相比，可以看到用户邮件列表的活跃度，Flink 比 Spark 更高一筹。此外，在开发者代码提交次数与 Github 的用户访问量中，Flink 在 Apache 所有项目中排名第二，在大数据项目中 Flink 都是排名第一。



从 2019 年 4 月至今，Flink 社区发布了 5 个版本，每一个版本都有更多的 Commit 与贡献者参与到其中。

二、为什么要学习 Apache Flink

(一) 大数据处理的实时化趋势



随着网络迅速发展，大数据的处理呈现出非常明显的实时化趋势。

如上图所示，我们列举了一些现实生活中常见的场景。如春晚的直播有一个实时大屏，双 11 购物节也有实时成交额的统计和媒体汇报。城市大脑可以实时监测交通，银行可以实时进行风控监测。当我们打开淘宝、天猫等应用软件时，它都会根据用户不同的习惯进行实时个性化推荐。从以上例子我们可以看到，实时化就是当下大数据处理的趋势。

(二) Flink 已成为国内外实时计算事实标准



在实时化的大趋势底下，Flink 已成为国内外实时计算事实标准。

如上图所示，目前国内外许多公司都在使用 Flink，国际公司有 Netflix、eBay，LinkedIn 等，国内有阿里巴巴、腾讯、美团、小米、快手等大型互联网公司。

(三) 流计算引擎的演进



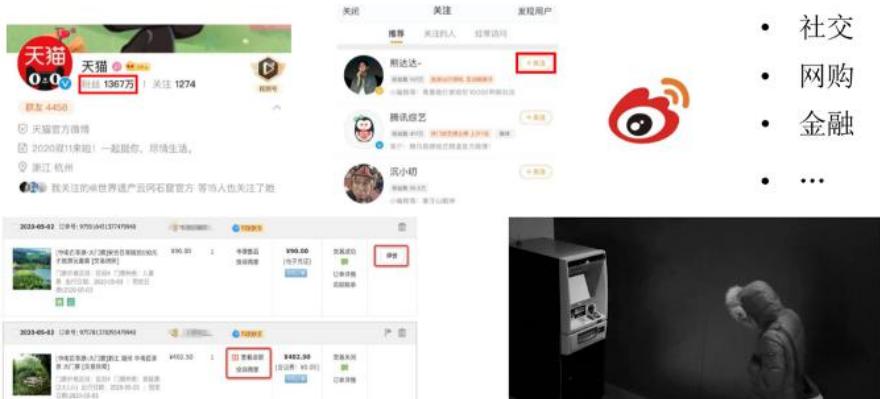
流计算引擎进行了很多代的演进，第一代流计算引擎 Apache Storm 是一个纯流的设计，延迟非常的低，但是它的问题也比较明显，即没有办法避免消息的重复处理，从而导致数据正确性有一定的问题。

Spark Streaming 是第二代流计算引擎，解决了流计算语义正确性的问题，但是它的设计理念是以批为核心，最大的问题是延迟比较高，只能做到 10 秒级别的延迟，端到端无法实现秒以内的延迟。

Flink 是第三代流计算引擎，也是最新一代的流计算引擎。它既可以保证低延迟，同时又可以保证消息的一致性语义，对于内置状态的管理，也极大降低了应用程序的复杂度。

三、Apache Flink 典型应用场景

(一) 事件驱动型应用



第一类应用场景是事件驱动型应用。

事件驱动表示一个事件会触发另一个或者是很多个后续的事件，然后这一系列事件会形成一些信息，基于这些信息需要做一定的处理。

在社交场景下，以微博为例，当我们点击了一个关注之后，被关注人的粉丝数就会发生变化。之后如果被关注的人发了一条微博，关注他的粉丝也会收到消息通知，这是一个典型的事件驱动。

另外，在网购的场景底下，如用户给商品做评价，这些评价一方面会影响店铺的星级，另外一方面有恶意差评的检测。此外，用户通过点击信息流，也可以看到商品派送或其他状态，这些都可能触发后续的一系列事件。

还有金融反欺诈的场景，诈骗者通过短信诈骗，然后在取款机窃取别人的钱财。在这种场景底下，我们通过摄像头拍摄后，迅速反应识别出来，然后对犯罪的行为进行相应的处理。这也是一个典型的事件驱动型应用。



总结一下，事件驱动型应用是一类具有状态的应用，会根据事件流中的事件触发计算、更新状态或进行外部系统操作。事件驱动型应用常见于实时计算业务中，比如：实时推荐，金融反欺诈，实时规则预警等。

(二) 数据分析型应用



第二类典型应用场景是数据分析型应用，如双 11 成交额实时汇总，包括 PV、UV 的统计。

包括上方图中所示，是 Apache 开源软件在全世界不同地区的一个下载量，其实也是一个信息的汇总。

还包括一些营销大屏，销量的升降，营销策略的结果进行环比、同比的比较，这些背后都涉及到大量信息实时的分析和聚合，这些都是 Flink 非常典型的使用场景。

40亿 消息处理峰值
每秒处理的数据条数
达到40亿



数据体量 7TB
每秒处理的数据量
达到7TB

58万 订单创建峰值
每秒创建的订单数
达到58万

计算规模 150万
集群内计算总规模
超过150万核

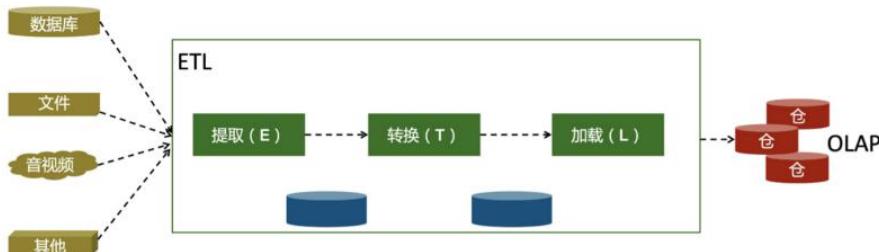
如上图所示，以双 11 为例，在 2020 年天猫双 11 购物节，阿里基于 Flink 的实时计算平台每秒处理的消息数达到了 40 亿条，数据体量达到 7TB，订单创建数达到 58 万/秒，计算规模也超过了 150 万核。

可以看到，这些应用的场景体量很大且对于实时性要求非常高，这也是 Apache Flink 非常擅长的场景。

(三) 数据管道型应用 (ETL)

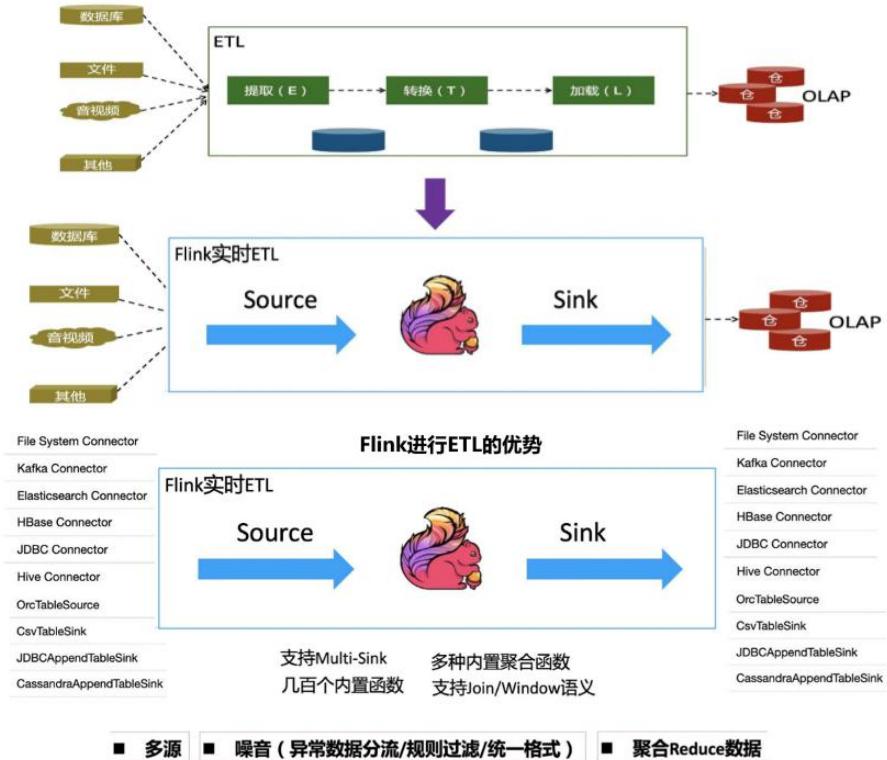
Apache Flink 擅长的第三类场景为数据管道型应用，即 ETL。

ETL (Extract-Transform-Load) 是从数据源抽取/转换/加载/数据至目的端的过程。



传统的 ETL 使用离线处理，经常做的是小时级别或者天级别的 ETL。

但是，随着大数据处理呈现实时化趋势，我们也会有实时数仓的需求，要求在分钟级或者秒级就能够对数据进行更新，从而进行及时的查询，能够看到实时的指标，然后做更实时的判断和分析。



在以上场景底下，Flink 能够最大限度地满足实时化的需求。

背后的原因主要有以下几个，一方面 Flink 有非常丰富的 Connector，支持多种数据源和数据 Sink，囊括了所有主流的存储系统。另外它也有一些非常通用的内置聚合函数来完成 ETL 程序的编写，因此 ETL 类型的应用也是它非常适合的应用场景。

四、Apache Flink 基本概念

(一) Flink 的核心概念

Flink 的核心概念主要有四个：Event Streams、State、(Event) Time 和 Snapshots。

1.1 Event Streams

即事件流，事件流可以是实时的也可以是历史的。Flink 是基于流的，但它不止能处理流，也能处理批，而流和批的输入都是事件流，差别在于实时与批量。

1.2 State

Flink 擅长处理有状态的计算。通常的复杂业务逻辑都是有状态的，它不仅要处理单一的事件，而且需要记录一系列历史的信息，然后进行计算或者判断。

1.3 (Event) Time

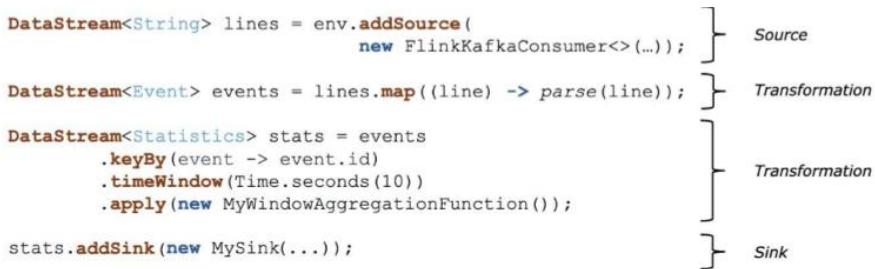
最主要处理的问题是数据乱序的时候，一致性如何保证。

1.4 Snapshots

实现了数据的快照、故障的恢复，保证数据一致性和作业的升级迁移等。

(二) Flink 作业描述和逻辑拓扑

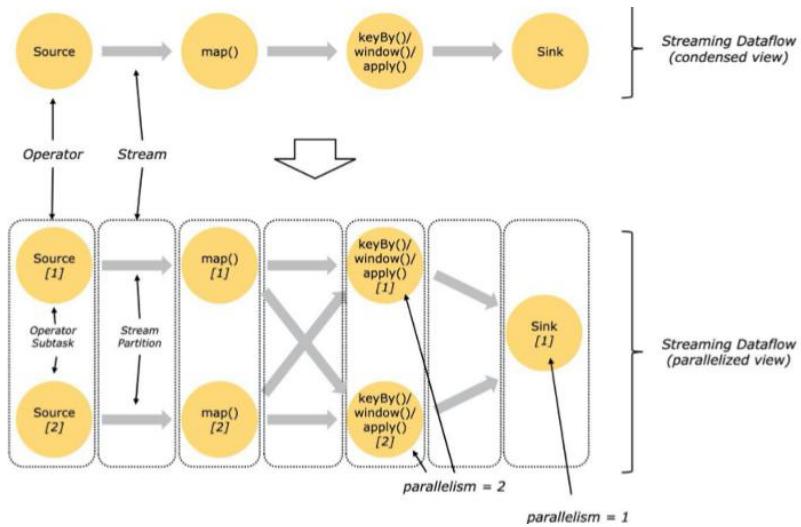
接下来我们来具体的去看一下 Flink 的作业描述和逻辑拓扑。



如上方所示，代码是一个简单的 Flink 作业描述。它首先定义了一个 Kafka Source，说明数据源是来自于 Kafka 消息队列，然后解析 Kafka 里每一条数据。解析完成后，下发的数据我们会按照事件的 ID 进行 KeyBy，每个分组每 10 秒钟进行一次窗口的聚合。聚合处理完之后，消息会写到自定义的 Sink。以上是一个简单的作业描述，这个作业描述会映射到一个直观的逻辑拓扑。

可以看到逻辑拓扑里面有 4 个称为算子或者是运算的单元，分别是 Source、Map、KeyBy/Window/Apply、Sink，我们把逻辑拓扑称为 Streaming Dataflow。

(三) Flink 物理拓扑

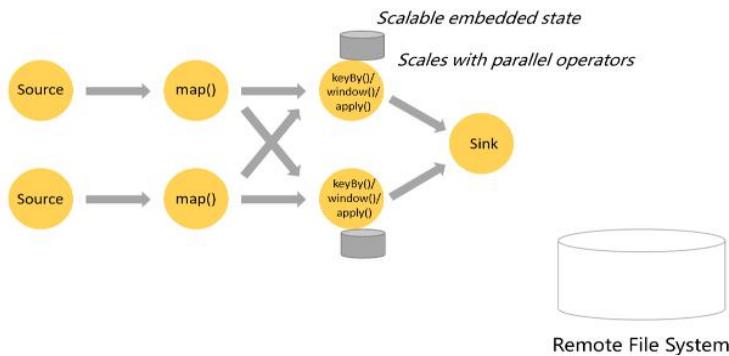


逻辑拓扑对应物理拓扑，它的每一个算子都可以并发进行处理，进行负载均衡与处理加速等。

大数据的处理基本上都是分布式的，每一个算子都可以有不同的并发度。有 Key By 关键字的时候，会按照 key 来对数据进行分组，所以在 KeyBy 前面的算子处理完之后，数据会进行一个 Shuffle 并发送到下一个算子里面。上图代表了示例对应的物理拓扑。

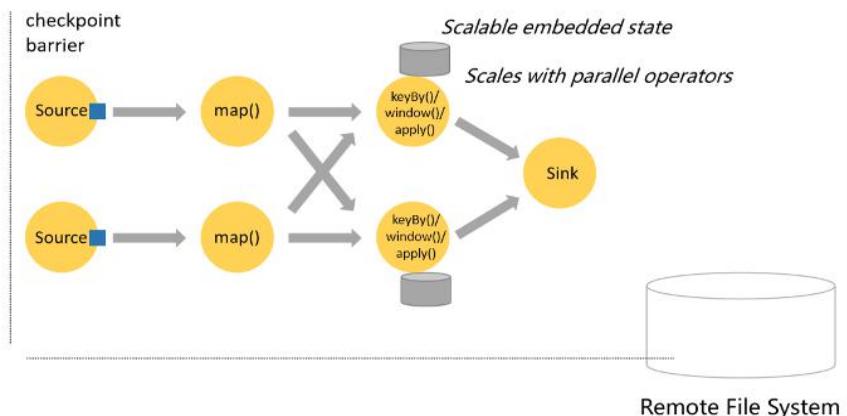
(四) Flink 状态管理和快照

接下来我们看一下 Flink 里面的状态管理和快照。



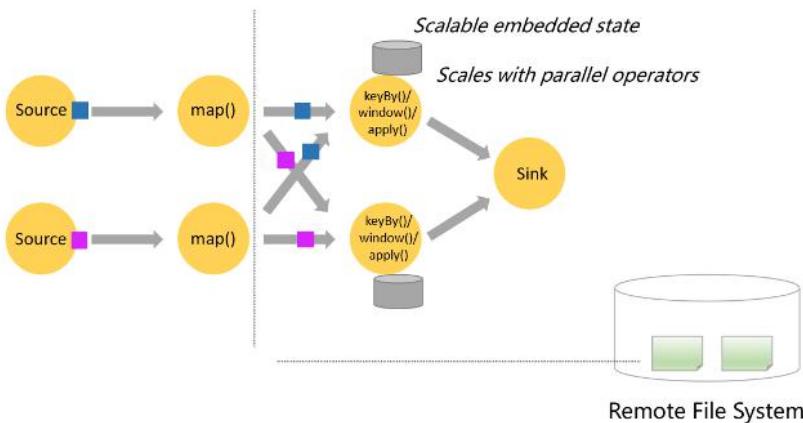
在进行 Window 的聚合逻辑时，每隔 10 秒会对数据进行聚合函数的处理。这 10 秒内的数据需要先存储起来，待时间窗口触发时进行处理。这些状态数据会以嵌入式存储的形式存储在本地。这里的嵌入式存储既可以是进程的内存里，也可以是类似 RocksDB 的持久化 KV 存储，两者最主要的差别是处理速度与容量。

此外，这些有状态算子的每一个并发都会有一个本地的存储，因此它的状态数据本身可以跟随算子的并发度进行动态的扩缩容，从而可以通过增加并发处理很大的数据量。

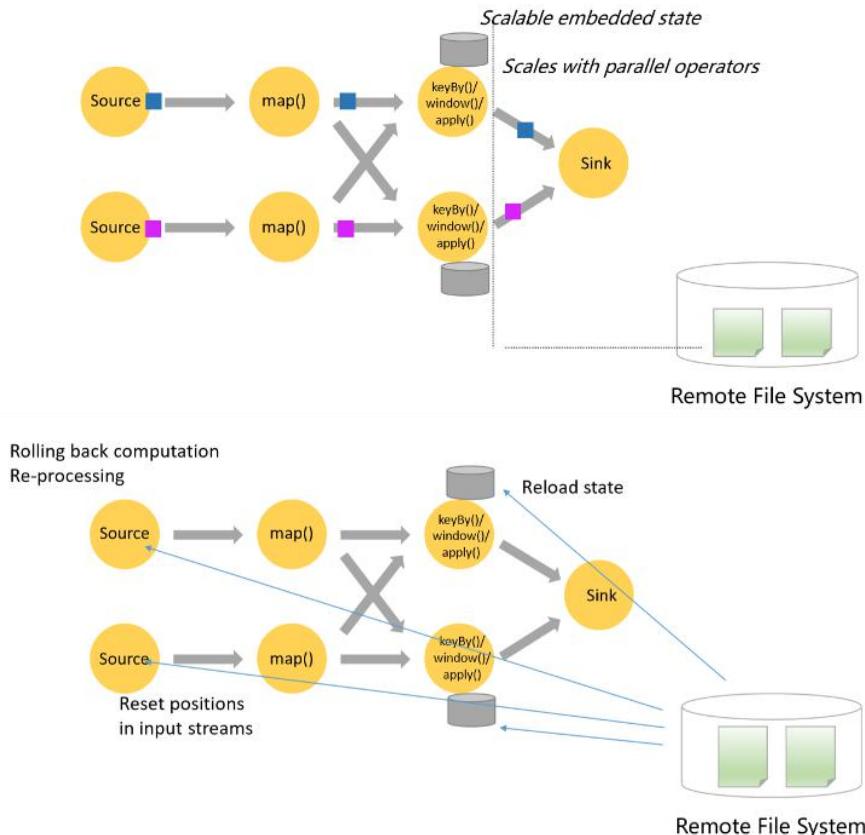


另一方面，作业在很多情况下有可能会失败。失败之后重新去运行时，我们如何保证数据的一致性？

Flink 基于 Chandy-Lamport 算法，会把分布式的每一个节点的状态保存到分布式文件系统里面作为 Checkpoint（检查点），过程大致如下。首先，从数据源端开始注入 Checkpoint Barrier，它是一种比较特殊的消息。

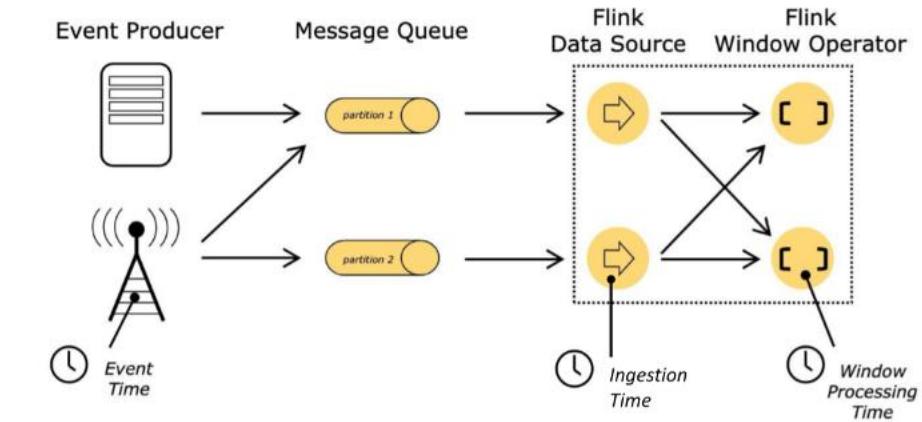


然后它会跟普通的事件一样随着数据流去流动，当 Barrier 到达算子之后，这个算子会把它当前的本地状态进行快照保存，当 Barrier 流动到 Sink，所有的状态都保存完整了之后，它就形成一个全局的快照。



这样当作业失败之后，就可以通过远程文件系统里面保存的 Checkpoint 来进行回滚：先把 Source 回滚到 Checkpoint 记录的 offset，然后把有状态节点当时的状况回滚到对应的时间点，进行重新计算。这样既可以不用从头开始计算，又能保证数据语义的一致性。

(五) Flink 中的时间定义



Flink 里另一个很重要的定义是 Event Time。

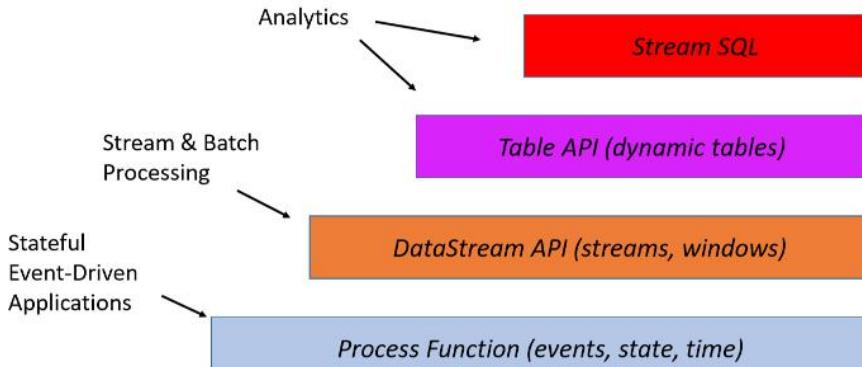
在 Flink 里有三种不同的时间，Event Time 指事件发生的时间，Ingestion Time 指事件到达 Flink 数据源的时间，或者说进入到 Flink 处理框架的时间，Processing Time 指处理时间，即到达算子当前的时间，这三个之间有什么区别呢？

在现实世界中，这个事件从发生到写入到系统里面，期间的间隔可能比较久。例如在地铁里面信号较弱时，如果我们在微博进行转发、评论、点赞等操作，由于网络的原因，这些操作可能要等我们出了地铁后才能完成，因此可能有些先发生的事件会后到达系统。而 Event Time 能够更真实地反映事件发生的时间点，因此在很多场景下，我们用 Event Time 作为事件发生的时间。

但是在这种情况底下，由于存在的延迟，所以在窗口需要花费较长的时间等待它的到来，端到端的延迟可能较大。

我们还需要处理乱序的问题，如果用 Processing Time 当做事件时间的话，处理较快，延迟较低，但是无法反映真实事件发生的情况。因此在真实的开发应用时，需要根据应用的特点做相应的取舍。

(六) Flink API



Flink 可分成 4 个层次的 API，最底层的 API 是可以自定义的 Process Function，对一些最基本的元素，如时间、状态等，进行细节的处理，实现自己的逻辑。

再往上一层是 DataStream API，它可以做流和批的处理，另外一方面它是逻辑的表达，有很多 Flink 内置的函数，方便用户编写程序。

最上层的 API 是 Table API 和 Stream SQL，这是一个非常上层的表达形式，非常简洁，我们接下来分别举例说明。

6.1 Process Function

可以看到，在 `processElement` 里边，能够对这个事件、状态进行自定义逻辑的处理。另外，我们可以注册一个 `timer`，并且自定义当 `timer` 被触发或时间到达的时候，到底要进行哪些处理，是一个非常精细的底层控制。

6.2 DataStream API

`DataStream API` 是作业的描述，可以看到它有很多内置的函数，如 `Map`、`keyBy`、`timeWindow`、`sum` 等。这也有些我们刚才自定义的 `ProcessFunction`，如 `MyAggregationFunction`。

6.3 Table API & Stream SQL

同样的逻辑，如果用 `Table API` 和 `Stream SQL` 描述的话，它就更加地直观。数据分析师不需要了解底层的细节，可以用一种描述式的语言去写逻辑。有关 `Table API` 和 `Stream SQL` 方面的内容，会在第 5 课进行详细的介绍。

(七) Flink 运行时架构

Flink 运行时的架构主要有三个角色。

第一个是客户端，客户端会提交它的应用程序，如果它是一个 `SQL` 程序，还会进行 `SQL` 优化器的优化，然后生成对应的 `JobGraph`。客户端会把 `obGraph` 提交到 `JobManager`，可以认为这是整个作业的主控节点。

JobManager 会拉起一系列的 TaskManager 作为工作节点，工作节点之间会按照作业拓扑进行串联，还有相应计算逻辑的处理，JobManager 主要是进行一些控制流的处理。

(八) Flink 物理部署

最后我们来看一下 Flink 能部署哪些环境。

首先，它可以通过手动的方式作业提交到 YARN, Mesos 以及 Standalone 集群上。另外，它也可以通过镜像的方式提交到 K8s 云原生的环境中。

目前，Flink 在许多物理环境中均能进行部署。

Stream Processing with Apache Flink

作者：崔星灿

Apache Flink Committer

本篇内容包含三部分展开介绍 Stream Processing with Apache Flink：

- 并行处理和编程范式
- DataStream API 概览及简单应用
- Flink 中的状态和时间

一、并行处理和编程范式

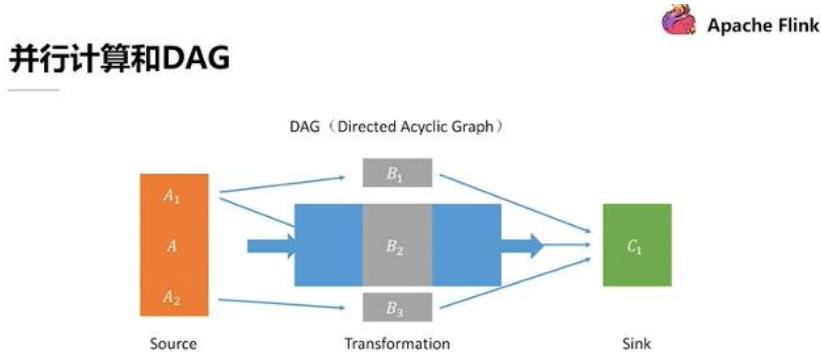
众所周知，对于计算密集型或数据密集型这样需要计算量比较大的工作，并行计算或分而治之是解决这一类问题非常有效的手段。在这个手段中比较关键的部分是，如何对一个已有任务的划分，或者说如何对计算资源进行合理分配。

举例说明，上学期间老师有时会找同学来协助批阅考试试卷。假如卷子里面一共有 ABC 三个题，那么同学可能会有如下分工协作方式。

方式一：将所有试卷的三个题分别交给不同的人来批阅。这种方式，每个批阅的同学批自己负责的题目后就可以把试卷传给下一个批阅同学，从而形成一种流水线的工作效果。但是这种流水线的协作方式会随着同学数量的增加而难以继续扩展。

方式二：分工方式一的扩展，同一题目允许多个同学来共同批阅，比如 A 题目由两个同学共同批阅，B 题目由三个同学批阅，C 题目只由一个同学批阅。这时候我们就需要考虑怎样进一步的对计算任务做划分。比如，可以把全部同学分成三组，第一组负责 A 题目，第二个组负责 B 题目第三个组负责 C。第一个组的同学可以再次在组内进行分工，比如 A 组里第一个同学批一半的卷子，第二个同学批另一半卷子。他们分别批完了之后，再将自己手里的试卷传递给下一个组。

像上述按照试卷内题目进行划分，以及讲试卷本身进行划分，就是所谓的计算的并行性和数据并行性。



Apache Flink Community China | 极客训练营

我们可以用上面有向无环图来表示这种并行性。

在图中，批阅 A 题目的同学，假设还承担了一些额外任务，比如把试卷从老师的办公室拿到批阅试卷的地点；负责 C 题的同学也有额外任务，就是等所有同学把试卷批完后，进行总分的统计和记录上交的工作。据此，可以把图中所有的节

点划分为三个类别。第一个类别是 Source，它们负责获取数据（拿试卷）；第二类是数据处理节点，它们大多时候不需要和外部系统打交道；最后一个类别负责将整个计算逻辑写到某个外部系统（统分并上交记录）。这三类节点分别就是 Source 节点、Transformation 节点和 Sink 节点。DAG 图中，节点表示计算，节点之间的连线代表计算之间的依赖。

(一) 关于编程的一些内容

The diagram illustrates the comparison between Imperative and Declarative programming styles using Apache Flink code examples.

命令式编程 (Imperative Programming):

```
List<Integer> data = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
public static int imperative() {
    List<Integer> tempList = new ArrayList<>();
    for (int v : data) {
        tempList.add(v * 2);
    }
    int result = 0;
    for (int v : tempList) {
        result += v;
    }
    return result;
}
```

声明式编程 (Declarative Programming):

```
public static int declarative() {
    return data.stream().mapToInt(v -> v * 2).sum();
}
SELECT SUM(2 * value) FROM data
```

Apache Flink Community China | 极客训练营

假设有一个数据集，其中包含 1~10 十个数字，如果把每一个数字都乘以 2 并做累计求和操作（如上图所示）怎么操作呢？办法有很多。

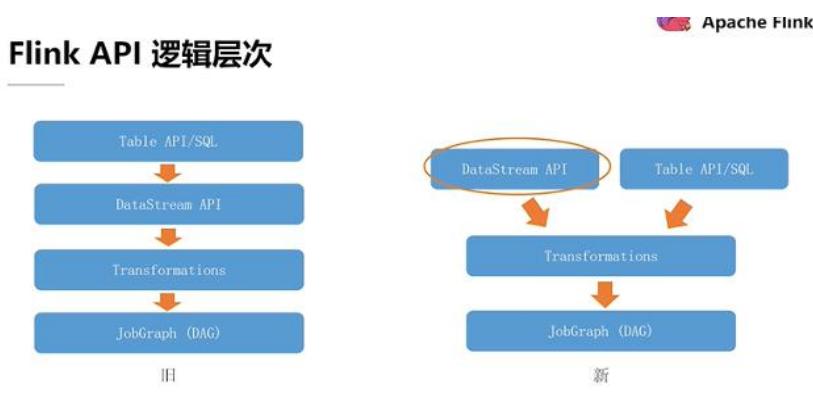
如果用编程来解决有两个角度：第一种是采取命令式编程方式，一步一步的相当于告诉机器应该怎样生成一些数据结构，怎样的用这些数据结构去存储一些临时的中间结果，怎样把这些中间结果再转换成为最终的结果，相当于一步一步告诉机器如何去做；第二种是声明的方式，声明式编程里通常只需要告诉机器去

完成怎样的任务，而不需要像命令式那样详细传递。例如我们可以把原有的数据集转化成一个 Stream，然后再把 Stream 转化成一个 Int 类型的 Stream，在此过程中，把每一个数字都乘 2，最后再调用 Sum 方法，就可以获得所有数字的和。

声明式编程语言的代码更简洁，而简洁的开发方式，正是计算引擎追求的效果。所以在 Flink 里所有与任务编写相关的 API，都是偏向声明式的。

二、DataStream API 概览及简单应用

在详细介绍 DataStream API 之前，我们先来看一下 Flink API 的逻辑层次。



在旧版本的 Flink 里，它的 API 层次遵循上图左侧这样四层的关系。最上层表示我们可以用比较高级的 API，或者说声明程度更高的 Table API 以及 SQL 的方式来编写逻辑。所有 SQL 和 Table API 编写的内容都会被 Flink 内部翻译和优化成一个用 DataStream API 实现的程序。再往下一层，DataStream API 的程序会

被表示成为一系列 Transformation，最终 Transformation 会被翻译成 JobGraph (即上文介绍的 DAG)。

而在较新版本的 Flink 里发生了一些改变，主要的改变体现在 Table API 和 SQL 这一层上。它不再会被翻译成 DataStream API 的程序，而是直接到达底层 Transformation 一层。换句话说，DataStream API 和 Table API 这两者的关系，从一个下层和上层的关系变为了一个平级的关系，这样流程的简化，会相应地带来一些查询优化方面的好处。

接下来我们用一个简单的 DataStream API 程序作为示例来介绍，还是上文乘 2 再求和的需求。

DataStream API 示例

```
public static void dataStream() throws Exception {
    //1. 获取运行环境
    StreamExecutionEnvironment e = StreamExecutionEnvironment.getExecutionEnvironment();
    //2. 设置Source读取数据
    DataStream<Integer> source = e.addSource(
        new FromElementsFunction<>(Types.INT.createSerializer(e.getConfig()), data), Types.INT);
    //3. 对数据进行一系列转换
    DataStream<Integer> ds = source.map(v -> v * 2).keyBy(value -> 1).sum(0);
    //4. 将数据写入Sink
    ds.addSink(new PrintSinkFunction<>());
    //5. 提交执行
    e.execute();
}
```

Apache Flink Community China | 极客训练营

如果用 Flink 表示，它的基本代码如上图所示。看上去比单机的示例要稍微的复杂一点，我们一步一步来分解看。

首先，用 Flink 实现任何功能，一定要获取一个相应的运行环境，也就是 Stream Execution Environment；

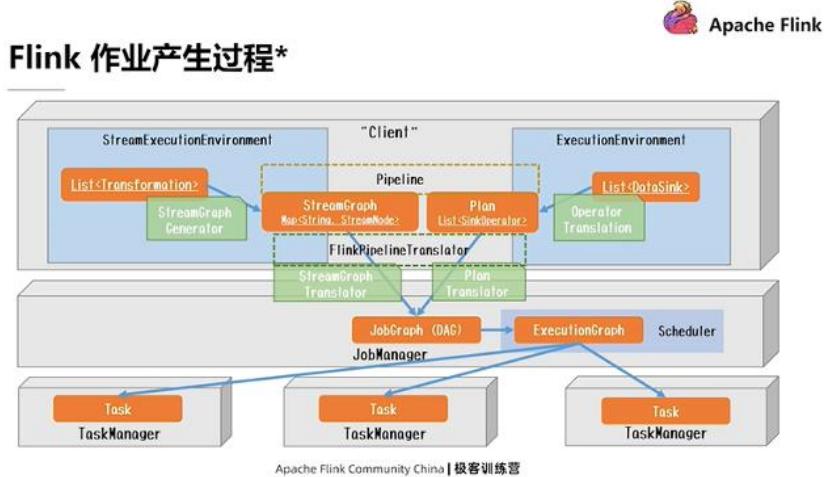
其次，在获取环境后，可以调用环境的 add Source 方法，来为逻辑添加一个最初始数据源的输入；设置完数据源后可以拿到数据源的引用，也就是 Data Source 对象；

最后，可以调用一系列的转换方法来对 Data Source 中的数据进行转化。

这种转化如图所示，就是把每个数字都 $\times 2$ ，随后为了求和我们必须利用 keyBy 对数据进行分组。传入的常数表示把所有的数据都分到一组里边，最后再对这个组里边的所有数据，按照第一个字段进行累加，最终得到结果。在得到结果后，不能简单的像单机程序那样把它输出，而是需要在整个逻辑里面加一个的 Sink 节点，把所有的数据写到目标位置。上述工作完成后，要去调用 Environment 里面 Execute 方法，把所有上面编写的逻辑统一提交到远程或者本地的一个集群上执行。

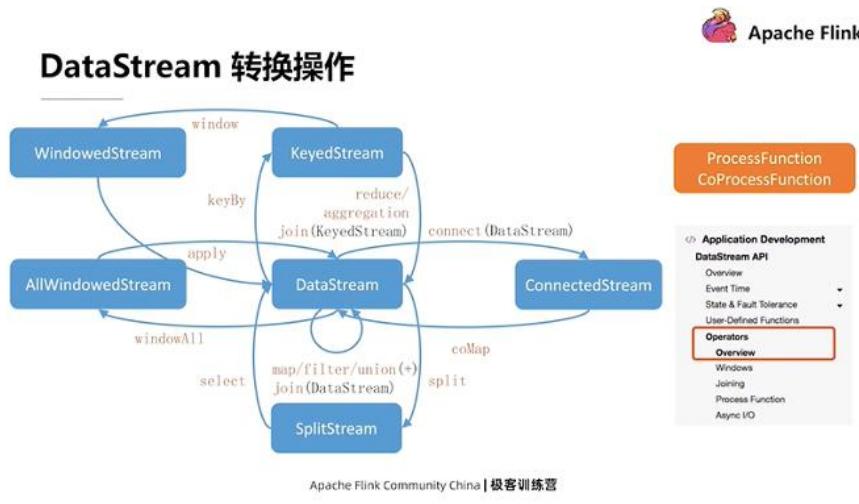
Flink DataStream API 编写程序和单机程序最大的不同就在于，它前几步的过程都不会触发数据的计算，而像在绘制一个 DAG 图。等整个逻辑的 DAG 图绘制完毕之后，就可以通过 Execute 方法，把整个的图作为一个整体，提交到集群上去执行。

介绍到这里，就把 Flink DataStream API 和 DAG 图联系在一起了。事实上，Flink 任务具体的产生过程比上面描述的要复杂得多，它要经过一步步转化和优化等，下图展示了 Flink 作业的具体生成过程。



(一) DataStream API 里提供的转换操作

就像上文在示例代码中展示的，每一个 DataStream 对象，在被调用相应方法的时候，都会产生一个新的转换。相应的，底层会生成一个新的算子，这个算子会被添加到现有逻辑的 DAG 图中。相当于添加一条连线来指向现有 DAG 图的最后一个节点。所有的这些 API 在调动它的时候都会产生一个新的对象，然后可以在新的对象上去继续调用它的转换方法。就是像这种链式的方式，一步一步把这个 DAG 图给画出来。



上述解释涉及到了一些高阶函数思想。每去调用 DataStream 上的一个转换时，都需要给它传递的一个参数。换句话说，转换决定了你想对这个数据进行怎样的操作，而实际传递的包在算子里面的函数决定了转换操作具体要怎样完成。

上图中，除了左边列出来的 API，Flink DataStream API 里面还有两个非常重要的功能，它们是 **ProcessFunction** 以及 **CoProcessFunction**。这两个函数是作为最底层的处理逻辑提供给用户使用的。上图所有左侧蓝色涉及的转换，理论上来讲都可以用底层的 **ProcessFunction** 和 **CoProcessFunction** 去完成。

(二) 关于数据分区

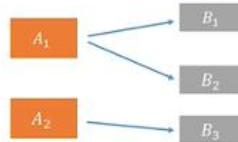
数据分区是指在传统的批处理中对数据 Shuffle 的操作。如果把扑克牌想成数据，传统批处理里的 Shuffle 操作就相当于理牌的过程。一般情况下在抓牌过程

中，我们都会把牌理顺排列好，相同的数字还要放在一起。这样做最大的好处是，出牌时可以一下子找到想出的牌。Shuffle 是传统的批处理的方式。因为流处理所有的数据都是动态来的，所以理牌的过程或者说处理数据，进行分组或分区的过程，也是在线来完成的。

数据分区 (Shuffle)



批处理 - Shuffle



流处理 - Partition

例如上图右侧所示，上游有两个算子 A 的处理实例，下游是三个算子 B 处理实例。这里展示的流处理等价于 Shuffle 的操作被称为数据分区或数据路由。它用来表示 A 处理完数据后，要把结果发到下游 B 的哪个处理实例上。

(三) Flink 里提供的分区策略

图 X 是 Flink 提供的分区策略。需要注意的是，DataStream 调用 keyBy 方法后，可以把整个数据按照一个 Key 值进行分区。但要严格来讲，其实 keyBy 并不算 是底层物理分区策略，而是一种转换操作，因为从 API 角度来看，它会把 DataStream 转化成 KeyedDataStream 的类型，而这两者所支持的操作也有所不同。



分区策略

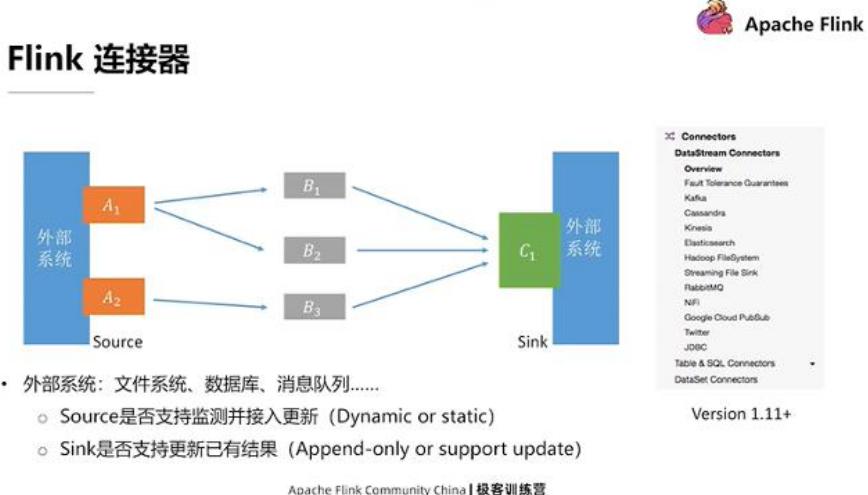
类型	描述
<code>dataStream.keyBy()</code>	按照Key值分区
<code>dataStream.global()</code>	全部发往第1个实例
<code>dataStream.broadcast()</code>	广播
<code>dataStream.forward()</code>	上下游并行度一样时一对一发送
<code>dataStream.shuffle()</code>	随机均匀分配
<code>dataStream.rebalance()</code>	Round-Robin (轮流分配)
<code>dataStream.rescale()</code>	Local Round-Robin (本地轮流分配)
<code>dataStream.partitionCustom()</code>	自定义单播

所有这些分区策略里，稍微难理解的可能是 Rescale。Rescale 涉及到上下游数据本地性的问题，它和传统的 Rebalance，即 Round-Pobin，轮流分配类似。区别在于 Rescale 是它会尽量避免数据跨网络的传输。

如果所有上述的分区策略都不适用的话，我们还可以自己调用 PartitionCustom 去自定义一个数据的分区。值得注意的是，它只是自定义的单播，即对每一个数据只能指定它一个下游所要发送的实例，而没有办法把它复制成多份发送到下游的多个实例中。

(四) Flink 支持的连接器

上文介绍过，图 X 里有两个关键的节点：A 节点，需要去连接外部系统，从外部系统把数据读取到 Flink 的处理集群里；C 节点，即 Sink 节点，它需要汇总处理完的结果，然后把这个结果写入到某个外部系统里。这里的外部系统可以是一个文件系统，也可以是一个数据库等。



Flink 里的计算逻辑可以没有数据输出，也就是说可以不把最终的数据写出到外部系统，因为 Flink 里面还有一个 State 的状态的概念。在中间计算的结果实际上是可以通过 State 暴露给外部系统，所以允许没有专门的 Sink。但每一个 Flink 应用都肯定有 Source，也就是说必须从某个地方把数据读进来，才能进行后续的处理。

关于 Source 和 Sink 两类连接器需要关注的点如下：

对于 Source 而言，我们往往比较关心是否支持持续监测并接入数据更新，然后把相应的更新数据再给传输到这个系统当中来。举例来说，Flink 对于文件有相应的 FileSystem 连接器，例如 CSV 文件。CSV 文件连接器在定义时，可以通过参数指定是否持续监测某个目录的文件变化，并接入更新后的文件。

对于 Sink 来讲，我们往往关心要写出的外部系统是否支持更新已经写出的结果。比如要把数据写到 Kafka 里，通常情况下数据写入是一种 Append-Only，即

不能修改已经写入系统里的记录（社区正在利用 Kafka Compaction 实现 Upsert Sink）；如果是写入数据库，那么通常可以支持利用主键对现有数据进行更新。

以上两个特性，决定了 Flink 里连接器是面向静态数据还是面向动态的数据的关键点。

提醒，上面截图是 Flink 1.11 版本之后的文档，连接器在 Flink 1.11 版本里有较大重构。另外，关于 Table、SQL、API 这个层面的连接器，比起 DataStream 层面的连接器，会承担更多的任务。比如是否支持一些谓词或投影操作的下推等等。这些功能可以帮助提高数据处理的整体性能。

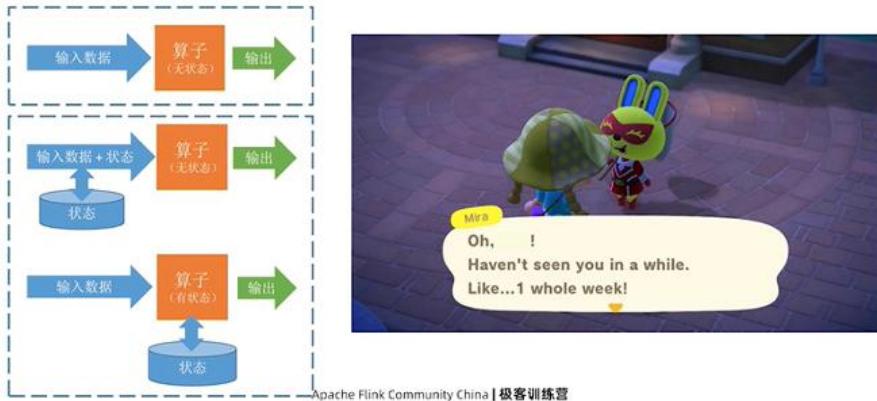
三、Flink 中的状态和时间

如果想要深入地了解 DataStream API，状态和时间是必须掌握的要点。

所有的计算都可以简单地分为无状态计算和有状态计算。无状态计算相对而言比较容易。假设这里有个加法算子，每进来一组数据，都把它们全部加起来，然后把结果输出去，有点纯函数的味道。纯函数指的是每一次计算结果只和输入数据有关，之前的计算或者外部状态对它不会产生任何影响。



有状态的计算



这里我们主要讲一下 Flink 里边的有状态计算。用捡树枝的小游戏来举例。这个游戏在我看来做的非常好的一点是它自己记录了非常多的状态，比如几天没上线，然后再去和里面的 NPC 对话的时候，它就会告诉你已经有好久没有上线了。换句话说，它会把之前上线的时间作为一种状态给记录下来，在生成与 NPC 对话的时候，是会受到这个状态的影响。

实现这种有状态的计算，要做的一点就是把之前的状态记录下来，然后再把这个状态注入到新的一次计算中，具体实现方式也有下面两种：

第一种，把状态数据进入算子之前就给提取出来，然后把这个状态数据和输入数据合并在一起，再把它们同时输入到算子中，得到一个输出。这种方式是被用在 Spark 的 Structured Streaming 里边。其好处是是可以重用已有的无状态算子。

第二种，是 Flink 现在的方法，就是算子本身是有状态的，算子在每一次到新数据之后做计算的时候，同时考虑新输入数据和已有的状态对计算过程的影响，最终把结果输出出去。

计算引擎也应该像上面提到的游戏一样变得越来越智能，可以自动学习数据中潜在的规律，然后来自适应地优化计算逻辑，保持较高的处理性能。

(一) Flink 的状态原语

Flink 的状态原语涉及如何通过代码使用 Flink 的状态。其基本思想是在编程的时候抛弃原生语言（例如 Java 或 Scala）提供的数据容器，把它们更换为 Flink 里面的状态原语。

作为对状态支持比较好的系统，Flink 内部提供了可以使用的很多种可选的状态原语。从大的角度看，所有状态原语可以分为 Keyed State 和 Operator State 两类。Operator State 应用相对比较少，我们在这里不展开介绍。下面重点看一下 Keyed State。



Keyed State，即分区状态。分区状态的好处是可以把已有状态按逻辑提供的分区分成不同的块。块内的计算和状态都是绑定在一起的，而不同的 Key 值之间的计算和状态的读写都是隔离的。对于每个 Key 值，只需要管理好自己的计算逻辑和状态就可以了，不需要去考虑其它 Key 值所对应的逻辑和状态。

Keyed State 可以进一步划分为下面的 5 类，它们分别是：

- 比较常用的：ValueState、ListState、MapState
- 不太常用的：ReducingState 和 AggregationState

Keyed State 只能在 RichFunction 中使用，RichFunction 与普通、传统的 Function 相比，最大的不同就是它有自己的生命周期。Key State 的使用方法分为以下四个步骤：

- 第一步，将 State 声明为 RichFunction 里的实例的变量
- 第二步，在 RichFunction 对应的 open 方法中，为 State 进行一个初始化的赋值操作。赋值操作要有两步：先创建一个 StateDescriptor，在创建中需要给 State 指定一个名称；然后再去调用 RichFunction 中的 getRuntimeContext().getState(...)，把刚刚定义的 StateDescriptor 传进去，就可以获取 State。

(提醒：如果此流式应用是第一次运行，那么获得的 State 会是空内容的；如果 State 是从某个中间段重启的，它会根据配置和之前保存的数据的基础上进行恢复。)



KeyedState 使用方法

1. 只能用于RichFunction
 2. 将State声明为实例变量
 3. 在open()方法中为State赋值
 - 创建一个StateDescriptor
 - 利用getRuntimeContext().getState(...)获得State
 4. 调用State的方法进行读写（例如：state.value()、state.update(..)）
-
- 第三步，得到 State 对象后，就可以在 RichFunction 里，对对应的 State 进行读写。如果是 ValueState，可以调用它的 Value 方法来获取对应值。Flink 框架会控制好所有状态的并发访问，并进行限制，所以用户不需要考虑并发的问题。

(二) Flink 的时间

时间也是 Flink 非常重要的一点，它和 State 是相辅相成的。总体来看 Flink 引擎里边提供的时间有两类：第一类是 Processing Time；第二类是 Event Time。Processing Time 表示的是真实世界的时间，Event Time 是数据当中包含的时间。数据在生成的过程当中会携带时间戳之类的字段，因为很多时候需要将数据里携带的时间戳作为参考，然后对数据进行分时间的处理。

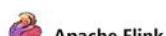


Flink 中的时间

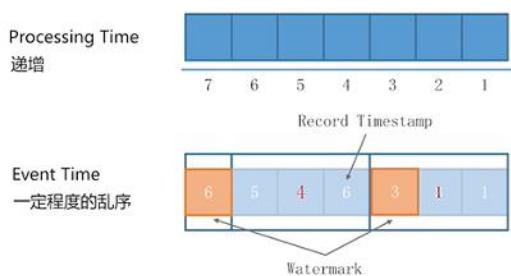
Processing Time	Event Time (Row Time)
真实世界的时间	数据世界的时间
处理数据节点的本地时间	记录携带的Timestamp
处理简单	处理复杂
结果不确定 (无法重现)	结果确定 (可重现)

Processing Time 处理起来相对简单，因为它不需要考虑乱序等问题；而 Event Time 处理起来相对复杂。由于 Processing Time 在使用时是直接调取系统的时间，考虑到多线程或分布式系统的不确定性，所以它每次运行的结果可能是不确定的；相反，因为 Event Time 时间戳是被写入每一条数据里的，所以在重放某个数据进行多次处理的时候，携带的这些时间戳不会改变，如果处理逻辑没有改变的话，最后的结果也是比较确定的。

Processing Time 和 Event Time 的区别。



Processing Time 和 Event Time



以上图的数据为例，按照 1~7 的时间来排列的。对于机器时间而言，每个机器的时间会单调增加。在这种情况下，用 Processing Time 获得的时间是完美的按照时间从小到大排序的数据。对于 Event Time 而言，由于延迟或分布式的一些原因，数据到来的顺序可能和它们真实产生的顺序有一定的出入，数据可能存在着一定程度的乱序。这时就要充分利用数据里边携带的时间戳，对数据进行一个粗粒度的划分。例如可以把数据分为三组，第一组里最小的时间是 1，第二组最小的时间是 4，第三组最小的时间是 7。这样划分之后，数据在组和组之间就是按从小到大的顺序排列好的。

怎样充分地把一定程度的乱序化解掉，让整个的系统看上去数据进来基本上是有顺序的？一种解决方案是在数据中间插入被称为 Watermark 的 meta 数据。在上图的例子中，前三个数据到来之后，假设再没有小于等于 3 的数据进来了，这时就可以插入一条 Watermark 3 到整个数据里，系统在看到 Watermark 3 时就知道，以后都不会有小于或等于 3 的数据过来了，这时它就可以放心大胆地进行自己的一些处理逻辑。

总结一下，Processing Time 在使用时，是一个严格递增的；而 Event Time 会存在一定的乱序，需要通过 Watermark 这种办法对乱序进行一定缓解。

从 API 的角度来看，怎样去分配 Timestamp 或生成 Watermark 也比较容易，有两种方式：

第一种，在 SourceFunction 当中调用内部提供的 collectWithTimestamp 方法，把包含时间戳的数据提取出来；还可以在 SourceFunction 中使用 emitWatermark 方法去产生一个 Watermark，然后插入到数据流中。



Timestamp分配和Watermark生成

- 在SourceFunction中产生
 - collectWithTimestamp(T element, long timestamp)
 - emitWatermark(Watermark mark)
- 在流程中指定
 - DataStream.assignTimestampsAndWatermarks(...)

定期生成	根据特殊记录生成
现实时间驱动	数据驱动
每隔一段时间调用生成方法	每一次分配Timestamp都会调用生成方法
实现AssignerWithPeriodicWatermarks	实现AssignerWithPunctuatedWatermarks

第二种, 如果不在 SourceFunction 中可以调用 DataStream.assignTimestampsAndWatermarks 这个方法, 同时传入两类 Watermark 生成器:

第一类是定期生成, 相当在环境里通过配置一个值, 比如每隔多长时间 (指真实时间) 系统会自动调用 Watermar 生成策略。

第二类是根据特殊记录生成, 如果遇到一些特殊数据, 可以采取 AssignWithPunctuatedWatermarks 这个方法来进行时间戳和 Watermark 的分配。

提醒: Flink 里内置了一些常用的 Assigner, 即 WatermarkAssigner。比如针对一个固定数据, 它会把这个数据对应的时间戳减去固定的时间作为一个 Watermark。关于 Timestamp 分配和 Watermark 生成接口, 在后续的版本可能会有一定的改动。注意, 新版本的 Flink 里面已经统一了上述两类生成器。

(三) 时间相关 API

Flink 在编写逻辑时会用到的与时间相关的 API，下图总结了 Event Time 和 Processing Time 相对应的 API。



时间相关API

目标	Event Time	Processing Time
获取记录事件	context.getTimestamp()或从数据字段中获取	timerService().currentProcessingTime()
获取“watermark”	timerService().currentWatermark()	timerService().currentProcessingTime()
注册定时器	timerService.registerEventTimeTimer()	timerService.registerProcessingTimeTimer()

在应用逻辑里通过接口支持可以完成三件事：

第一， 获取记录的时间。Event Time 可以调 context.getTimestamp，或在 SQL 算子内从数据字段中把对应的时间给提取出来。Processing Time 可以直接调 currentProcessingTime 完成调取，它的内部是直接调用了获取系统时间的静态方法来返回的值。

第二， 获取 Watermark。其实只有在 Event Time 里才有 Watermark 的概念，而 Processing Time 里是没有的。但在 Processing Time 中非要把某个东西当成 Watermark，其实就是数据时间本身。也就是说第一次调用 timerService.currentProcessingTime 方法之后获取的值。这个值既是当前记录的这个时间，也是当前的 Watermark 值，因为时间总是往前流动的，第一次调用了这个值后，第二次调用时这个值肯定不会再比第一次值还小。

第三，注册定时器。定时器的作用是清理。比如需要对一个 cache 在未来某个时间进行清理工作。既然清理工作应该发生在未来的某个时间点，那么可以调用 timerService.eventTimeTimer 或 ProcessingTimeTimer 方法注册定时器，再在整个方法里添加一个对定时器回调的处理逻辑。当对应的 Event Time 或者 Processing Time 的时间超过了定时器设置时间，它就会调用方法自己编写定时器的回调逻辑。

以上就是关于 StreamProcess with Apache Flink 的介绍，下一篇内容将着重介绍 Flink Runtime Architecture。

Flink Runtime Architecture

作者：朱翥（长耕）

Apache Flink PMC, 阿里巴巴技术专家

本文由 Apache Flink PMC 及 Committer 朱翥分享，主要介绍 Flink Runtime 的底层架构。本篇文章包含四部分：

- Runtime 总览
- 作业的控制中心—Jobmaster
- 任务的运行容器—TaskExecutor
- 资源的管理中心—ResourceManager

一、Runtime 总览

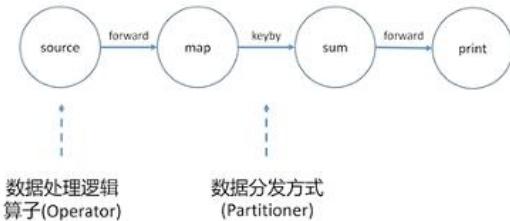
众所周知 Flink 是分布式的数据处理框架，用户的业务逻辑会以 Job 的形式提交给 Flink 集群。Flink Runtime 作为 Flink 引擎，负责让这些作业能够跑起来并正常完结。这些作业既可以是流计算作业，也可以是批处理作业，既可以跑在裸机上，也可以在 Flink 集群上跑，Flink Runtime 必须支持所有类型的作业，以及不同条件下运行的作业。

(一) 作业的表达

要执行作业，首先要理解作业是如何在 Flink 中进行表达的。

Runtime 总览 – 作业的表达

```
env.addSource(new StreamingWordInput())
    .map(word -> new Tuple2(word, 1))
    .keyBy(r -> r.f0).sum(1)
    .print();
```



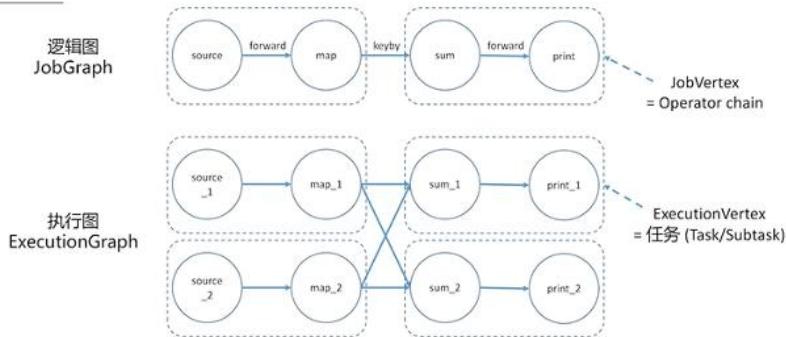
用户通过 API 的方式写一个作业，例如上图左侧 StreamWordInput 的示例，它可以不断的输出一个个单词；下面的 Map 操作负责把单词映射成一个二元组；再接一个 keyBy，使相同的 word 的二元组都被分配在一起，然后 sum 将它们计数，最后打印出来。

左侧的作业对应着右边的逻辑拓扑(StreamGraph)。这个拓扑中有 4 个节点，分别是 source、map、sum 和 print。这些是数据处理逻辑，又称之为算子；节点之间的线条对应着数据的分发方式，影响着数据以什么样的方式分发给下游。举例来说，map 到 sum 之间是 keyBy，意味着 map 产出的数据，同一个 key 的数据都必须分发到同一个下游。

有了 StreamGraph 后，Flink Runtime 会进一步的把它翻译成 JobGraph。JobGraph 和 StreamGraph 的区别是，JobGraph 会把一些节点 chain 起来，形成 Operator chain。Chain 条件是需要两个算子的并发度是一样的，并且它们的数据交换方式是一对一的。形成的 Operator chain，又称为 JobVertex。



Runtime 总览 – 作业的表达



Operator chain 的意义是能够减少一些不必要的数据交换，这样 chain 的 operator 都是在同一个地方进行执行。在作业实际执行过程中，逻辑图会进一步被翻译成执行图 — ExecutionGraph。执行图是逻辑图并发层面的视图，如上图所示，下面的执行图就是上面逻辑图所有算子并发都为 2 的表达。

为什么上图中的 map 和 sum 不能嵌起来？因为它们的数据是涉及到多个下游算子的，并非一对一的数据交换方式。逻辑图 JobVertex 中的一个节点，会对应着并发数个执行节点 ExecutionVertex，节点对应着一个个任务，这些任务最后会作为实体部署到 Worker 节点上，并执行实际的数据处理业务逻辑。

(二) 分布式架构

Flink 作为分布式数据处理框架，它有一套分布式的架构，主要分为三块：Client、Master 和 Worker 节点。

Master 是 Flink 集群的主控中心，它可以有一个到多个 JobMaster，每个 JobMaster 对应一个作业，而这些 JobMaster 由一个叫 Dispatcher 的控件统一管理。Master 节点中还有一个 ResourceManager 进行资源管理。ResourceManager 管理着所有 Worker 节点，它同时服务于所有作业。此外 Master 节点中还有一个 Rest Server，它会用于响应各种 Client 端来的 Rest 请求，Client 端包括 Web 端以及命令行的客户端，它可以发起的请求包括提交作业、查询作业的状态和停止作业等等。作业会通过执行图被划分成一个个的任务，这些任务最后都会在 Worker 节点中进行执行。Worker 就是 TaskExecutor，它们是任务执行的容器。

作业执行的核心组件有三个，分别是 JobMaster、TaskExecutor 和 ResourceManager：JobMaster 用于管理作业；TaskExecutor 用于执行各个任务；ResourceManager 用于管理资源，并服务于 JobMaster 的资源请求。

二、JobMaster：作业的控制中心

JobMaster 的主要职责包括作业生命周期的管理、任务的调度、出错恢复、状态查询和分布式状态快照。

分布式状态快照包括 Checkpoint 和 Savepoint，其中 Checkpoint 主要是为出错恢复服务的，而 Savepoint 主要是用于作业的维护，包括升级和迁移等等。

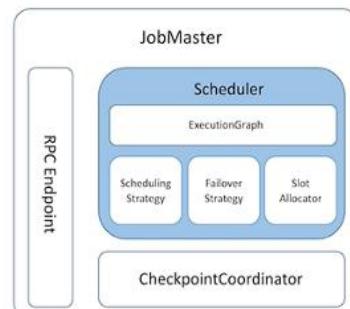
分布式快照是由 CheckpointCoordinator 组件来进行触发和管理的。



JobMaster - 作业的控制中心

主要职责：

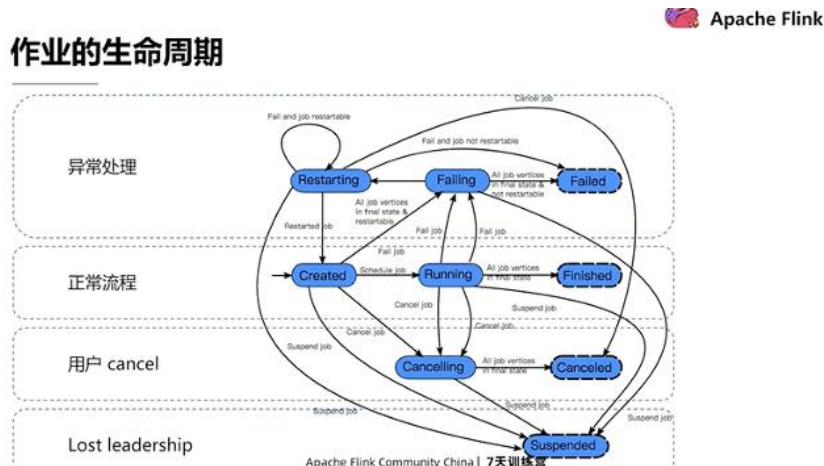
1. 作业生命周期管理
2. 任务调度
3. 出错恢复
4. 作业状态查询
5. 分布式状态快照



JobMaster 中的核心组件是 Scheduler，无论是作业的生命周期管理、作业的状态维护，还是任务的调度以及出错恢复，都是由 Schedule 来负责的。

(一) 作业的生命周期管理

作业的生命周期的状态，作业所有可能的状态迁移都在下图展示出来了。



正常流程下作业会有三种状态，分别是 Created、Running 和 Finished。一个作业开始是处于 Created 的状态，当这个作业被开始调度就开始进入 Running 状态并开始调度任务，等到所有的任务都成功结束了，这个作业就走到 Finished 的状态，并汇报最终结果，然后退出。

然而，一个作业在执行过程中可能会遇到一些问题，因此作业也会有异常处理的状态。作业执行过程中如果出现作业级别错误，整个作业会进到 Failing 状态，然后 Cancel 所有任务。等到所有任务都进入最终状态后，包括 Failed、Canceled、Finished，再去 check 出错的异常。如果异常是不可恢复的，那么整个作业会走到 Failed 状态并退出。如果异常是可恢复的，那么会走到 Restarting 状态，来尝试重启。如果重启的次数没有超过上限，作业会从 Created 状态重新进行调度；如果达到上限，作业会走到 Failed 状态并退出。（注：在 Flink 1.10 之后的版本中，当发生错误时，如果可以恢复，作业不会进入 Failing 状态而会直接进入 Restarting 状态，当所有任务都恢复正常后，作业会回到 Running 状态。

如果作业无法恢复，则作业会经由 Failing 状态最终进入 Failed 状态并结束。）

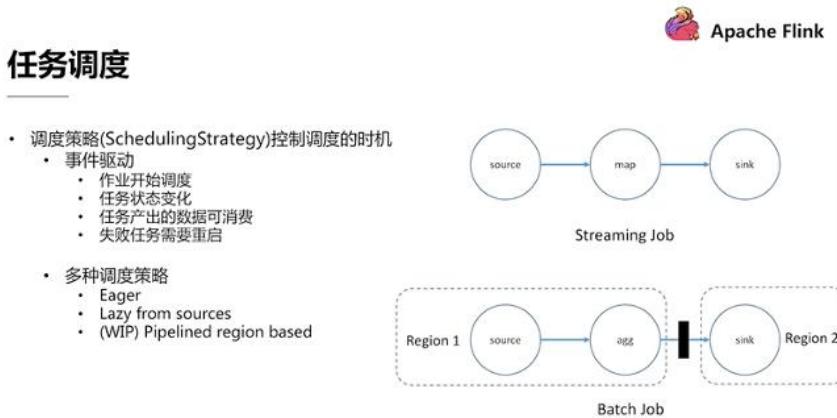
Cancelling 和 Canceled 两种状态只会在用户手动去 Cancel 作业的时候走到。当用户手动的在 Web UI 或通过 Flink command 探索作业的时候，Flink 会首先把状态转到 Cancel 里，然后 Cancel 所有任务，等所有任务都进入最终状态后，整个作业就会进入 Canceled 状态并退出。

Suspended 状态只会在配置了 high availability，并且当 JobMaster 丢掉 leadership 才会走到。这个状态只意味着这个 JobMaster 出现问题终止了。一般来说等到 JobMaster 重新拿到 leadership 之后，或是另外有一个 standby Master 拿

到 leadership 之后，会在拿到 leadership 的节点上重新启动起来。

(二) 任务调度

任务调度是 JobMaster 的核心职责之一。要调度任务，一个首要的问题就是决定什么时候去调度任务。任务调度时机是由调度策略（SchedulingStrategy）来控制的。这个策略是一个事件驱动的组件，它监听的事件包括：作业开始调度、任务的状态发生变化、任务产出的数据变成可消费以及失败的任务需要重启，通过监听这些事件，它能够比较灵活地来决定任务启动的时机。



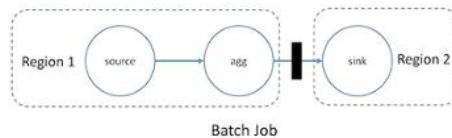
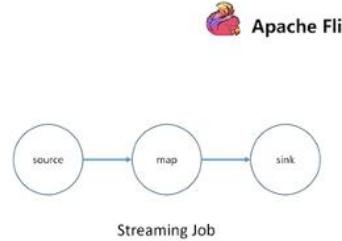
目前我们有多种不同的调度策略，分别是 Eager 和 Lazy from sources。EagerSchedulingStrategy 主要是服务于流式作业，它的策略是在作业开始调度时，直接启动所有的任务，这样做好处是可以降低调度时间。Lazy from sources 主要服务于批处理作业。它的策略是作业一开始只调度 Source 节点，等到有任何节点的输入数据可以被消费后，它才会被调起来。如下图所示，source 节点的

数据开始产出后，agg 节点才能被调起来，agg 节点结束后，sink 节点才能被调起来。

为什么 Batch 作业和 Streaming 作业会有不同的调度策略呢？是因为 Batch 作业里边存在 blocking shuffle 数据交换模式。在这种模式下，需要等上游完全产出所有数据后，下游才能去消费这部分数据集，如果预先把下游调起来的话，它只会在那空转浪费资源。相比 Eager 策略而言，对于批处理作业它能够节省一定量的资源。

任务调度

- 调度策略(SchedulingStrategy)控制调度的时机
 - 事件驱动
 - 作业开始调度
 - 任务状态变化
 - 任务产出的数据可消费
 - 失败任务需要重启
 - 多种调度策略
 - Eager
 - Lazy from sources
 - (WIP) Pipelined region based



目前还有一个正在开发中的叫 Pipelined region based 调度策略，这个策略比较类似于 Lazy from source 策略，差异在于前者是以 Pipelined region 为粒度调度任务的。

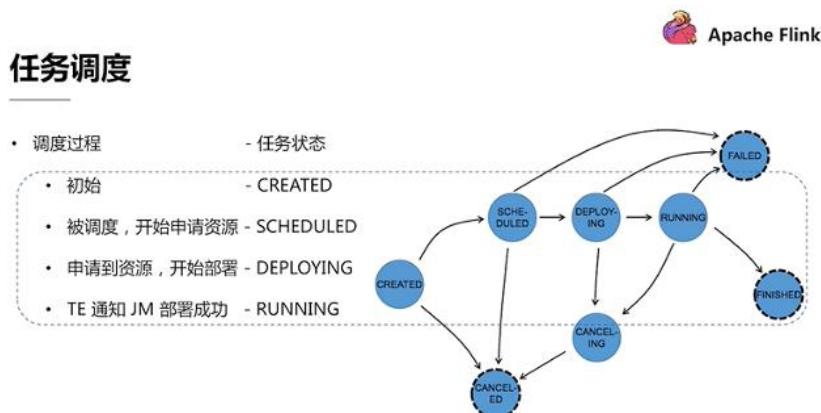
Pipelined region 是以 pipelined 相连的任务集合。Pipelined 边意味着上下游节点会流式的进行数据交换，即上游边写，下游就边读边消费。Pipelined region 调度的好处是可以一定程度上继承了 Eager 调度好处，能够节省调度花费的

时间，且让上下游任务并行起来。同时也保留了 Lazy from sources 避免不必要的资源浪费。通过把一部分 Task 整体调度，就能知道这部分需要同时运行的作业所需的资源量是多少，能够以此进行一些更深度的优化。

(注：从 Flink 1.11 开始，Pipelined region strategy 成为默认调度策略，同时服务于流和批作业。)

(三) 任务调度的过程

任务具有很多种不同状态，最初任务处在 Created 状态。当调度策略认为这个任务可以开始被调的时候，它会转到 Scheduled 状态，并开始申请资源，即 Slot。申请到 Slot 之后，它就转到 Deploying 状态来生成 Task 的描述，并部署到 worker 节点上，再之后 Task 就会在 worker 节点上启动起来。成功启动后，它会在 worker 节点上转到 running 状态并通知 JobMaster，然后在 JobMaster 端把任务的状态转到 running。



对于无限流的作业来说，转到 running 状态就是最终状态了；对于有限流的作业，一旦所有数据处理完了，任务还会转到 finished 状态，标志任务完成。当有异常发生时，任务也会转到 Failed 的状态，同时其它受到影响的任务可能会被 Cancel 掉并走到 Canceled 状态。

(四) 出错恢复

当有任务出现错误时，JobMaster 的策略或基本思路是，通过重启出错失败的任务以及可能受到影响的任务，来恢复作业的数据处理。这包含三个步骤：

- **第一步**，停止相关任务，包括出错失败任务和可能受其影响任务，失败任务可能已经是 FAILED 的，然后其它受影响任务会被 Cancel 最终进到 Canceled 状态；
- **第二步**，重置任务回 Created 状态；
- **第三步**，通知调度策略重新调度这些任务。

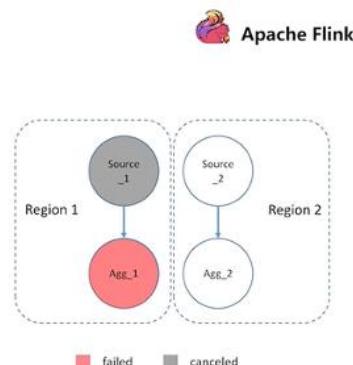
出错恢复

重启出错失败的任务以及可能受其影响的任务

- 停止相关任务 - FAILED/CANCELED
- 重置任务状态 - CREATED
- 通知调度策略重新调度

由出错恢复策略(FailoverStrategy)决定需要重启的任务

- RestartPipelinedRegionFailoverStrategy
- RestartAllFailoverStrategy
- 单点重启 ??



上文提及了可能受到影响的任务，那么什么样的任务可能受影响呢？这是由出错恢复策略（FailoverStrategy）来决定的。

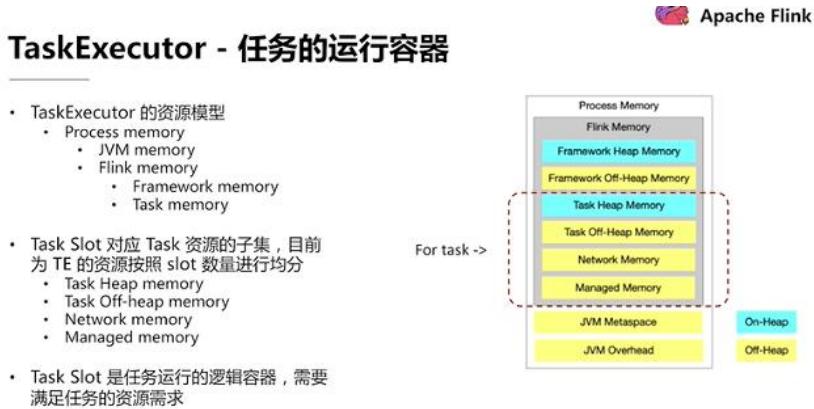
目前 Flink 默认的 FailoverStrategy 是 RestartPipelinedRegionFailoverStrategy。采用了这个策略后，如果一个 Task 失败了就会重启它所在的 region。这其实跟上文提及的 Pipelined 数据交换有关系。在 Pipelined 数据交换的节点之间，如果任意一个节点失败了，其相关联的其它节点也会跟着失败。这是为了防止出现数据的不一致。因此为了避免单个 Task 导致多次 Failover，一般的操作是在收到第一个 Task failed 时，就把其他的一起 cancel 掉，再一起重启。

RestartPipelinedRegion 策略除了重启失败任务所在的 Region 外，还会重启它的下游 Region。原因是任务的产出很多时候是非确定性的，比如说一个 record，分发到下游的第一个并发，重跑一次；分发到下游的第二个并发时，一旦这两个下游在不同 region 中，就可能会导致 record 丢失，甚至产生不一样的数据。为了避免这种情况，采用 PipelinedRegionFailoverStrategy 会重启失败任务所在的 Region 以及它的所有的下游 Region。

另外，还有一个 RestartAllFailoverStrategy 策略，它会在任意 Task fail 的时候，重启作业中的所有任务。一般情况，这个策略并不被经常用到，但是在一些特殊情况下，比如当任务失败，用户不希望局部运行而是希望所有任务都结束并整体进行恢复，可以用这个策略。

三、TaskExecutor：任务的运行器

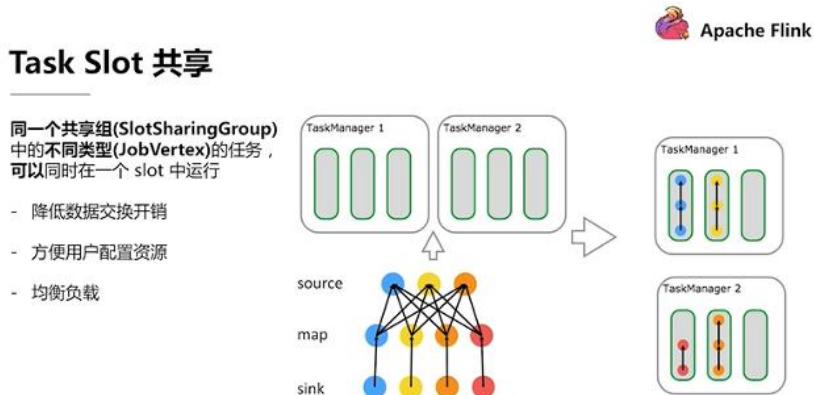
TaskExecutor 是任务的运行器，为了运行任务它具有各种各样的资源。如下图所示，这里主要介绍 memory 的资源。



所有内存资源都是可以单独配置的。TaskManager 也对它们的配置进行了分层的管理，最外层是 Process Memory，对应的是整个 TaskExecutor JVM 的总资源。这份内存又包含了 JVM 自身占有的内存以及 Flink 占有内存。而 Flink 占用内存又包含了框架占有的内存和任务的内存。

任务占用内存包括了 Task Heap Memory，即任务的 Java 对象占有的内存； Task Off-Heap Memory 一般用于 native 的第三方库； Network Memory 是用来创建 Network Buffer 用来服务于任务的输入和输出； Managed Memory 则是受管控的 Off-Heap Memory，它会被一些组件用到，比如算子和 StateBackend。这些 Task 资源会被它分成一个一个的 Slot，Slot 是任务运行的逻辑容器。当前，Slot 大小是直接把整个 TaskExecutor 的资源，按照 Slot 的数量进行均分得到的。

一个 Slot 里可以运行一个到多个任务，但是有一定约束，即同一个共享组中的不同类型的任务才可以同时在一个 Slot 中运行。一般来说，同一个 PipelinedRegion 中的任务都是在一个共享组中，流式作业的所有任务也都是在一个共享组中。不同类型指的是它们需要属于不同的 JobVertex。



如上图右侧示例，这是一个 source、map、sink 的作业。可以看到部署后，有三个 Slot 中都有三个任务，分别是 source、map、sum 各一份。而有一个 slot 中只有两个任务，就是因为 source 只有三个并发，没有更多并发可以部署进来。

进行 SlotSharing 第一个好处是，可以降低数据交换的开销。像 map、sink 之间是一对一的数据交换，实际上有物理数据交换的这些节点都被共享在了一块，这样可以使得它们的数据交换在内存中进行，比在网络中进行的开销更低。

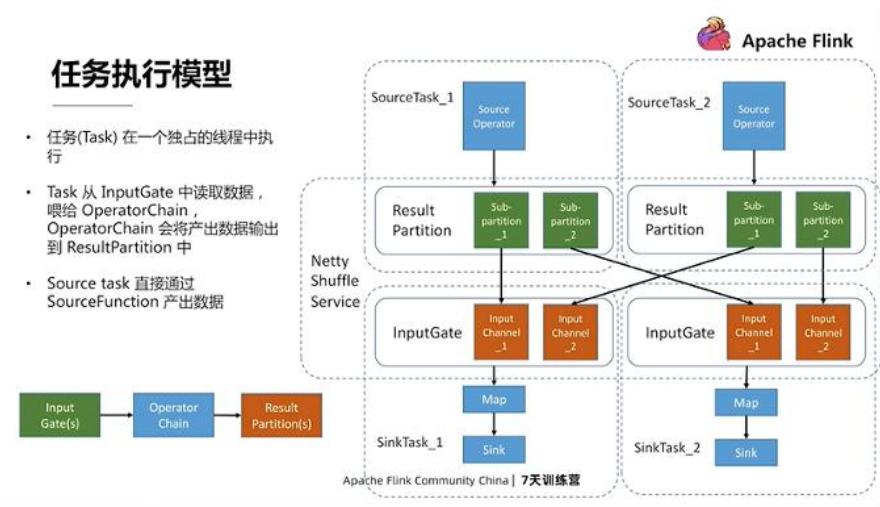
第二个好处是，方便用户配置资源。通过 SlotSharing，用户只需要配置 n 个 Slot 就可以保证一个 sum 作业总能跑起来。n 是最大算子的并发度。

第三个好处是，在各个算子并发度差异不大的情况下，提高负载均衡。这是因为每个 Slot 里边会有各种不同类型的算子各一份，这就避免某些负载重的算子全挤在同一个 TaskExecutor 中。

(一) 任务的执行模型

上文提到每个任务对应着一个 OperatorChain。一般来说每个 OperatorChain 都有自己的输入和输出，输入是 InputGate，输出是 ResultPartition。这些任务总会在一个独占的线程中执行，任务从 InputGate 中读取数据，将它喂给 OperatorChain，OperatorChain 进行业务逻辑的处理，最后会将产出的数据输出到 ResultPartition 中。

这地方有一个例外是 Source task，它不从 InputGate 中读取数据，而直接通过 SourceFunction 方式来产出数据。上游的 ResultPartition 和下游的 InputGate 会通过 Flink 的 ShuffleService 进行数据交换。ShuffleService 是一个插件，目前 Flink 默认是 NettyShuffleService，下游的 InputGate 会通过 Netty 来从上游的 ResultPartition 中获取数据。



ResultPartition 是由一个个的 SubPartition 组成的，每个 SubPartition 都对应着一个下游消费者并发。InputGate 也是由一个个的 InputChannel 组成的，每个不同的 InputChannel 都对应着一个上游并发。

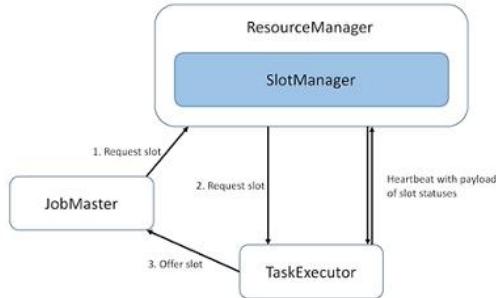
四、ResourceManager：资源的管理中心

ResourceManager 是 Flink 的资源管理中心。在前面我们有提到过 TaskExecutor 包含了各种各样的资源。而 ResourceManager，就管理着这些 TaskExecutor。新启动的 TaskExecutor，需要向 ResourceManager 进行注册，之后它里边的资源才能服务于作业的请求。

Apache Flink

ResourceManager – 资源的管理中心

- ResourceManager 管理 TaskExecutor
- SlotManager 管理 Slot
 - 通过 TE-RM 心跳更新 Slot 状态，心跳信息中包含 TE 中所有的 slot 状态
- Slot 申请流程
 - JM -> RM -> TE -> JM, 以 slot offer 的结果为准



ResourceManager 里边有个关键组件叫做 SlotManager，它管理着 Slot 的状态。这些 Slot 状态是通过 TaskExecutor 到 ResourceManager 之间的心跳跳来进行更新的，在心跳信息中包含了 TaskExecutor 中的所有 Slot 的状态。有了当前所有的 Slot 状态之后，ResourceManager 就可以服务于作业的资源申请。当 JobMaster 调度一个任务的时候，会向 ResourceManager 发起 Slot 请求。收到请求的 ResourceManager 会转交给 SlotManager，SlotManager 会去检查它里边的可用的 Slot 有没有符合请求条件的。如果说有的话，它就会向相应的 TaskExecutor 发起 Slot 申请。如果请求成功，TaskExecutor 会主动的向 JobMaster offer 这个 Slot。

之所以要这么绕一圈，是为了避免分布式带来的不一致的问题。像刚才我们有提到，SlotManager 中的 Slot 状态是通过心跳来进行更新的，所以存在一定的延迟。此外在整个 Slot 申请过程中，Slot 状态也是可能发生变化的。所以最终我们需要以 Slot offer，以及它的 ACK 来作为所有的申请的最终结果。

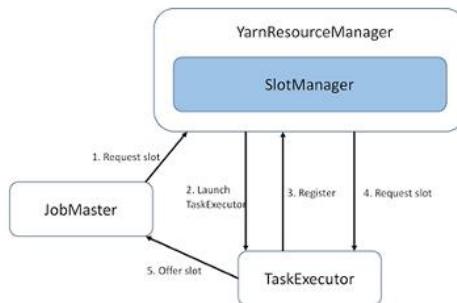
ResourceManager 有多种不同的实现，Standalone 模式下采用的 ResourceManager，是 StandaloneResourceManager，需要用户手动拉起 Worker 节点，这样就要求用户要先了解这个作业会需要多少总资源。

除此之外，还有一些会去自动申请资源的 ResourceManager，包括 YarnResourceManager，MesosResourceManager 和 KubernetesResourceManager。采用这些 ResourceManager 后，在不能满足的情况下，ResourceManager 会在 Slot 的请求过程自动拉起 Worker 节点。



ResourceManager 的多种实现

- StandaloneResourceManager
 - 手动拉起 Worker
- YarnResourceManager
- MesosResourceManager
- KubernetesResourceManager
 - Slot 申请过程中自动拉起 Worker



拿 YarnResourceManager 举个例子，JobMaster 去为某个任务请求一个 Slot。YarnResourceManager 将这个请求交给 SlotManager，SlotManager 发觉没有 Slot 符合申请的话会告知 YarnResourceManager，YarnResourceManager 就会向真正的外部的 YarnResourceManager 去请求一个 container，拿到 container 之后，它会启动一个 TaskExecutor，当 TaskExecutor 起来之后，它会注册到 ResourceManager 中，并去告知它可用的 Slot 信息。SlotManager 拿到

这个信息之后，就会尝试去满足当前 pending 的那些 SlotRequest。如果能够满足，JobMaster 就会去向 TaskExecutor 发起 Slot 请求，请求成功的话，TaskExecutor 就会向 JobMaster 去 offer 这个 Slot。这样用户就不需要在一开始去计算它作业的资源需求量是多少，而只需要保证单个 Slot 的大小，能够满足任务的执行了。

Fault-tolerance in Flink

作者：李钰

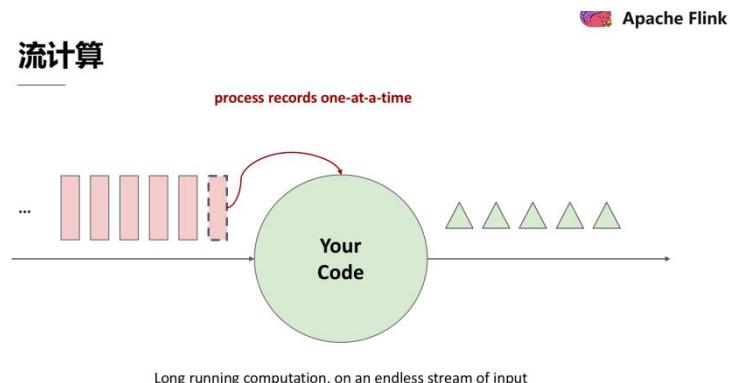
Apache Flink PMC , 阿里巴巴高级技术专家

本文由 Apache Flink PMC , 阿里巴巴高级技术专家李钰分享，主要介绍 Flink 的容错机制原理，内容大纲如下：

- 有状态的流计算
- 全局一致性快照
- Flink 的容错机制
- Flink 的状态管理

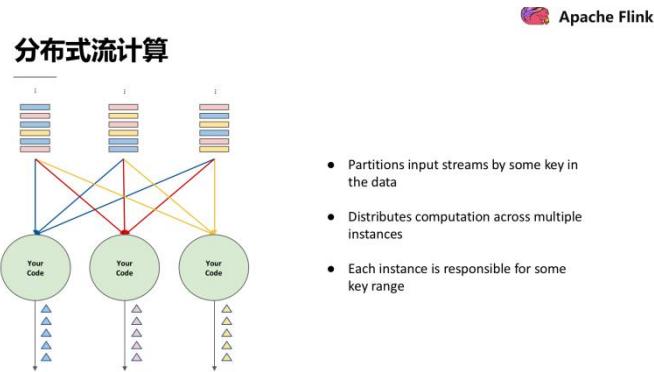
一、有状态的流计算

(一) 流计算



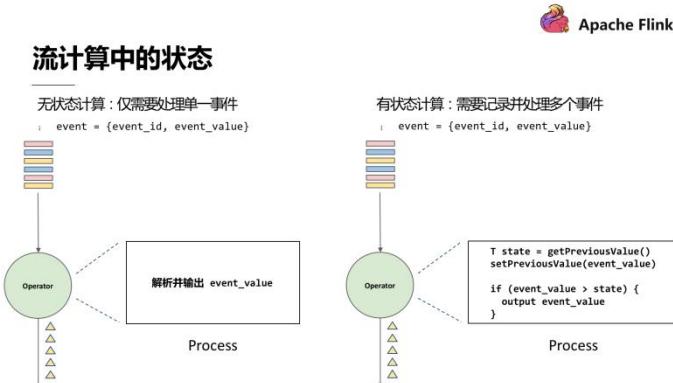
流计算是指有一个数据源可以持续不断地发送消息，同时有一个常驻程序运行代码，从数据源拿到一个消息后会进行处理，然后把结果输出到下游。

(二) 分布式流计算



分布式流计算是指把输入流以某种方式进行一个划分，再使用多个分布式实例对流进行处理。

(三) 流计算中的状态



计算可以分成有状态和无状态两种，无状态的计算只需要处理单一事件，有状态的计算需要记录并处理多个事件。

举个简单的例子。例如一个事件由事件 ID 和事件值两部分组成，如果处理逻辑是每拿到一个事件，都解析并输出它的事件值，那么这就是一个无状态的计算；相反，如果每拿到一个状态，解析它的值出来后，需要和前一个事件值进行比较，比前一个事件值大的时候才把它进行输出，这就是一个有状态的计算。



流计算中的状态



去重
记录所有的主键



窗口计算
已进入未触发的数据



机器学习/深度学习
训练的模型及参数



访问历史数据
记录历史数据

流计算中的状态有很多种。比如在去重的场景下，会记录所有的主键；又或者在窗口计算里，已经进入窗口还没触发的数据，这也是流计算的状态；在机器学习/深度学习场景里，训练的模型及参数数据都是流计算的状态。

二、全局一致性快照

全局一致性快照是可以用来给分布式系统做备份和故障恢复的机制。

(一) 全局快照

什么是全局快照



什么是全局快照



分布式应用

多个进程（领导人），分布在多个服务器上（多国）运行



应用间互相通信

消息在“管道”内进行传递



应用内部有处理逻辑和状态

处理数据并随时间产生变化



某一时刻的全局状态

包括各个进程的本地状态和传递中的消息

全局快照首先是一个分布式应用，它有多个进程分布在多个服务器上；其次，它在应用内部有自己的处理逻辑和状态；第三，应用间是可以互相通信的；第四，在这种分布式的应用，有内部状态，硬件可以通信的情况下，某一时刻的全局状态，就叫做全局的快照。

为什么需要全局快照



为什么需要全局快照



检查点 (checkpointing)

用于应用程序故障恢复



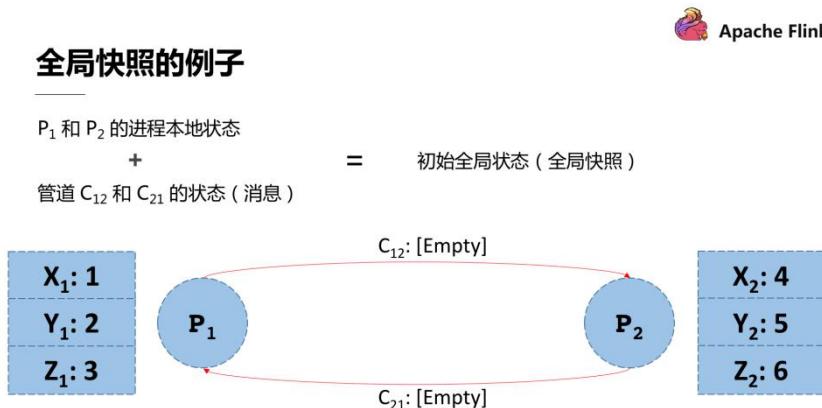
死锁检测

检查当前应用是否存在死锁而不影响程序运行

- 第一，用它来做检查点，可以定期对全局状态做备份，当应用程序故障时，就可以拿来恢复；
- 第二，做死锁检测，进行快照后当前的程序继续运行，然后可以对快照进行分析，看应用程序是不是存在死锁状态，如果是就可以进行相应的处理。

全局快照举例

下图为分布式系统中全局快照的示例。



P_1 和 P_2 是两个进程，它们之间有消息发送的管道，分别是 C_{12} 和 C_{21} 。对于 P_1 进程来说， C_{12} 是它发送消息的管道，称作 output channel； C_{21} 是它接收消息的管道，称作 input channel。

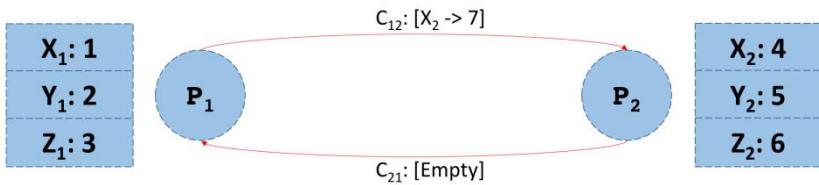
除了管道，每个进程都有一个本地的状态。比如说 P_1 和 P_2 每个进程的内存里都有 XYZ 三个变量和相应的值。那么 P_1 和 P_2 进程的本地状态和它们之间发送消息的管道状态，就是一个初始的全局状态，也可称为全局快照。



全局快照的例子

P_1 发送消息给 P_2 将其状态变量 X_2 的值从 4 更改为 7

这也组成一个全局快照



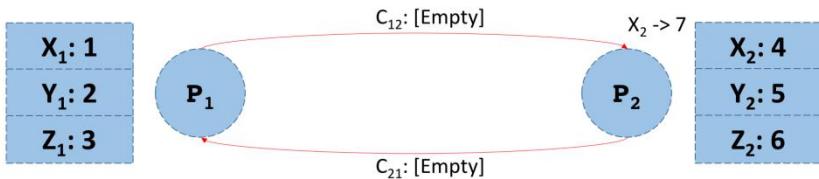
假设 P_1 给 P_2 发了一条消息，让 P_2 把 x 的状态值从 4 改为 7，但是这个消息在管道中，还没到达 P_2 。这个状态也是一个全局快照。



全局快照的例子

P_2 收到了 P_1 的消息

第 3 个全局快照



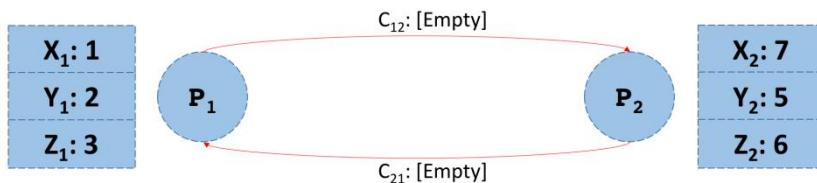
再接下来， P_2 收到了 P_1 的消息，但是还没有处理，这个状态也是一个全局快照。



全局快照的例子

P₂ 将本地变量从 4 更改为 7

第 4 个全局快照



最后接到消息的 P2 把本地的 X 的值从 4 改为 7，这也是一个全局快照。

所以当有事件发生的时候，全局的状态就会发生改变。事件包括进程发送消息、进程接收消息和进程修改自己的状态。

(二) 全局一致性快照



全局一致性快照

当事件发生时，全局的状态会发生改变，这里的事件包括：

- 进程发送消息
- 进程接收到消息
- 进程修改状态

a->b 代表在绝对时钟 (real time) 下 a happened before b, 则当一个全局快照满足下述条件时，我们称其为一个全局一致性快照：

- 如果 A->B 且 B 被包含在该快照中，则 A 也被包含在这个快照中

作业:

1. 分布式系统中的一致性: <https://zhuanlan.zhihu.com/p/55034347>
2. 分布式系统中的因果性、逻辑时钟、向量时钟: <https://zhuanlan.zhihu.com/p/57062155>
<http://lamport.azurewebsites.net/pubs/pubs.html#time-clocks>

假如说有两个事件， a 和 b ，在绝对时间下，如果 a 发生在 b 之前，且 b 被包含在快照当中，那么则 a 也被包含在快照当中。满足这个条件的全局快照，就称为全局一致性快照。

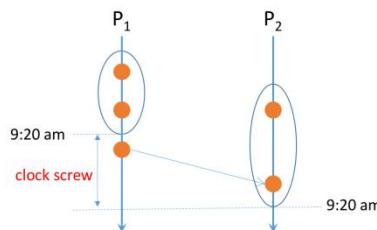
全局一致性快照的实现方法



全局一致性快照的实现方法

时钟同步？

✗ 时钟偏差会破坏全局一致性（NTP 存在毫秒级别的偏差）



全局同步？

✗ Stop-the-world，
影响应用运行

时钟同步并不能实现全局一致性快照；全局同步虽然可以实现，但是它的缺点也非常明显，它会让所有应用程序都停下来，会影响全局的性能。

(三) 异步全局一致性快照算法 – Chandy-Lamport

异步全局一致性快照算法 Chandy-Lamport 可以在不影响应用程序运行的前提下，实现全局一致性快照。

Chandy-Lamport 的系统要求有以下几点：

- 第一，不影响应用运行，也就是不影响收发消息，不需要停止应用程序；
- 第二，每个进程都可以记录本地状态；
- 第三，可以分布式地对已记录的状态进行收集；
- 第四，任意进程都可以发起快照。

同时，Chandy-Lamport 算法可以执行还有一个前提条件：消息有序且不重复，并且消息可靠性可保障。

Chandy-Lamport 算法流程



Chandy-Lamport 的算法流程主要分为三个部分：发起快照、分布式的执行快照和终止快照。

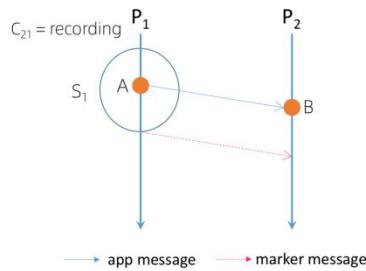
发起快照

任意进程都可以发起快照。如下图所示，当由 P1 发起快照的时候，第一步需要记录本地的状态，也就是对本地进行快照，然后立刻向它所有 output channel 发送一个 marker 消息，这中间是没有时间间隙的。marker 消息是一个特殊的消息，它不同于应用之间传递的消息。

Chandy-Lamport 算法流程

发起快照

- 记录本地状态（本地快照）
- 向 output channel 发送 marker 消息
- 开始记录所有 input channel 的消息



发出 Marker 消息后，P1 就会开始记录所有 input channel 的消息，也就是图示 C21 管道的消息。

分布式的执行快照

如下图，先假定当 P_i 接收到来自 C_{ki} 的 marker 消息，也就是 P_k 发给 P_i 的 marker 消息。可以分两种情况来看：

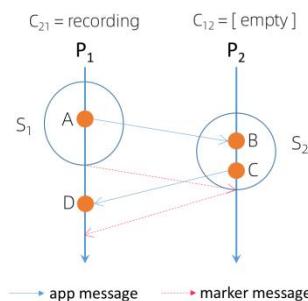


Chandy-Lamport 算法流程

分布式执行快照：当 P_i 接收到来自 C_{ki} 的 marker 消息（即 P_k 发给 P_i 的 marker）

如果这是 P_i 看到的第一个 marker 消息

- P_i 记录本地状态（本地快照）
- P_i 标记 C_{ki} 为空
- P_i 向所有 output channel 发送 marker 消息
- P_i 开始记录所有除 C_{ki} 之外的 input channel 消息
(即后续所有来自 C_{ki} 的消息不再包含进此次快照)



第一种情况：这个是 P_i 收到的第一个来自其它管道的 marker 消息，它会先记录一下本地的状态，再把 C_{12} 管道记为空，也就是说后续再从 P_1 发消息，就不包含在此次快照里了，与此同时立刻向它所有 output channel 发送 marker 消息。最后开始记录来自除 C_{ki} 之外的所有 input channel 的消息。

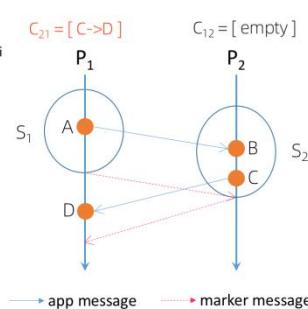


Chandy-Lamport 算法流程

分布式执行快照：当 P_i 接收到来自 C_{ki} 的 marker 消息（即 P_k 发给 P_i 的 marker）

如果此前 P_i 已经收到过 marker 消息

- P_i 停止记录 C_{ki} 的消息，同时将此前记录的所有 C_{ki} 收到的消息作为 C_{ki} 在本次快照中的最终状态



上面提到 C_{ki} 消息不包含在实时快照里，但是实时消息还是会发生，所以第二种情况是，如果此前 P_i 已经接收过 marker 消息，它会停止记录 C_{ki} 消息，同时会将此前记录的所有 C_{ki} 消息作为 C_{ki} 在本次快照中的最终状态来保存。

终止快照

终止快照的条件有两个：

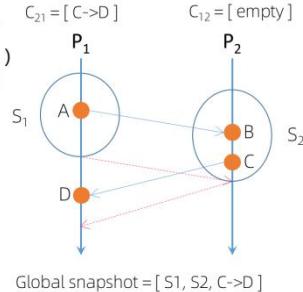
- 第一，所有进程都已经接收到 marker 消息，并记录在本地快照；
- 第二，所有进程都从它的 $n-1$ 个 input channel 里收到了 marker 消息，并记录了管道状态。

Chandy-Lamport 算法流程



终止快照

- 所有进程都已经接到了 marker 并记录了本地快照
- 所有进程都从 $N-1$ 个 input channel 里收到了 marker 并记录了这些 input channel 的状态（消息）
- 快照收集器（central server）可以开始收集每个部分的快照并形成全局一致性快照



当快照终止，快照收集器 (Central Server) 就开始收集每一个部分的快照去形成全局一致性快照了。

示例展示

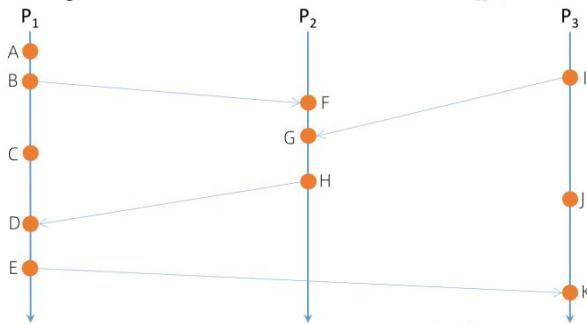
在下图的例子里，一些状态是在内部发生的，比如 A，它跟其它进程没有交互。

内部状态就是 P1 发给自己消息，可以将 A 认为是 $C_{11}=[A \rightarrow A]$ 。



Chandy-Lamport 算法 – 示例

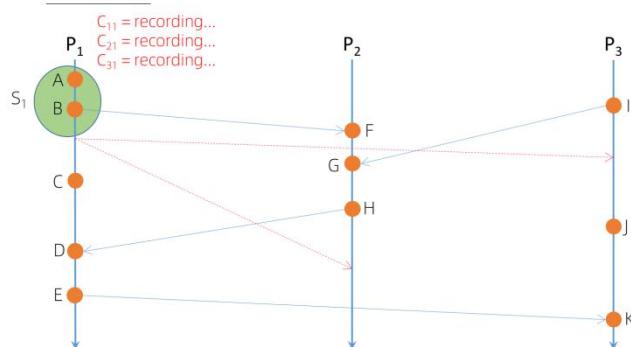
Extending: 内部状态认为是自己发给自己的消息，例如 A 认为是 $C_{11}=[A \rightarrow A]$



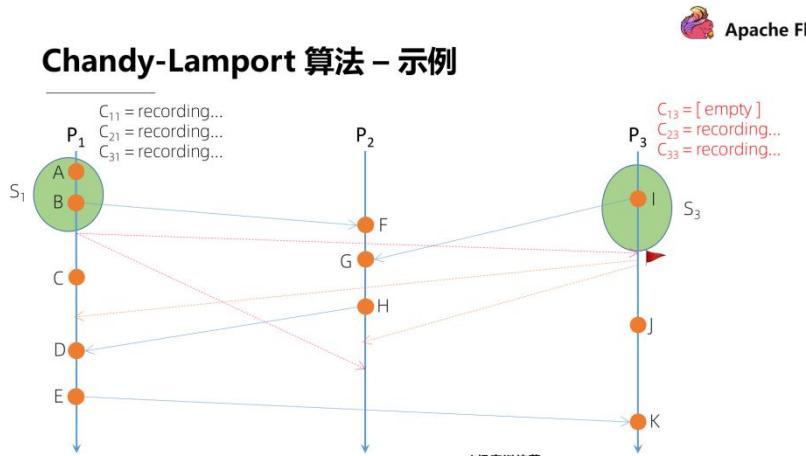
Chandy-Lamport 全局一致性快照的算法是怎么执行的呢？



Chandy-Lamport 算法 – 示例



假设从 p1 来发起快照，它发起快照时，首先对本地的状态进行快照，称之为 S1，然后立刻向它所有的 output channel，即 P2 和 P3，分别发 marker 消息，然后再去记录它所有 input channel 的消息，即来自 P2 和 P3 及自身的消息。

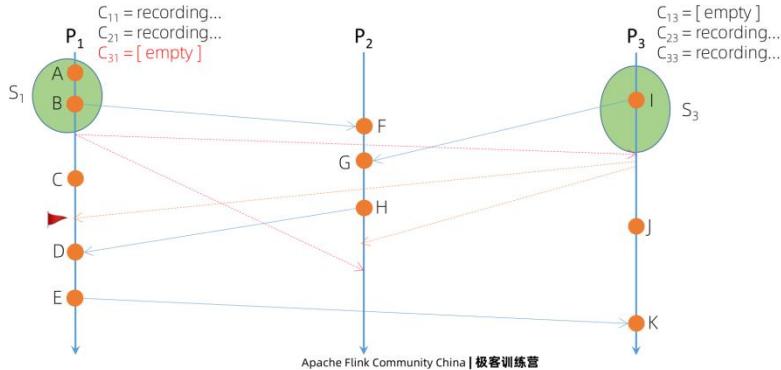


图例所示，纵轴是绝对时间，按照绝对时间来看，为什么 P3 和 P2 收到 marker 消息会有时间差呢？因为假如这是一个真实的物理环境里的分布式进程，不同节点之间的网络状况是不一样的，这种情况会导致消息送达时间存在差异。

P3 先收到 marker 消息，且是它接收到的第一个 marker 消息。接收到消息后，它首先会对本地状态进行快照，然后把 C13 管道的标记成 close，与此同时开始向它所有的 output channel 发送 marker 消息，最后它会把来自除了 C13 之外的所有 input channel 的消息开始进行记录。



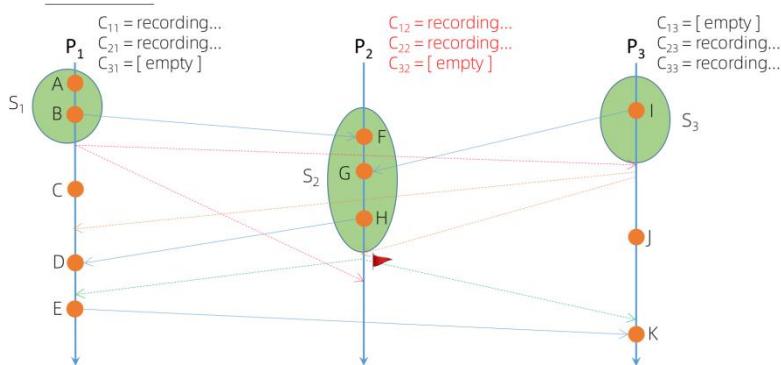
Chandy-Lamport 算法 – 示例



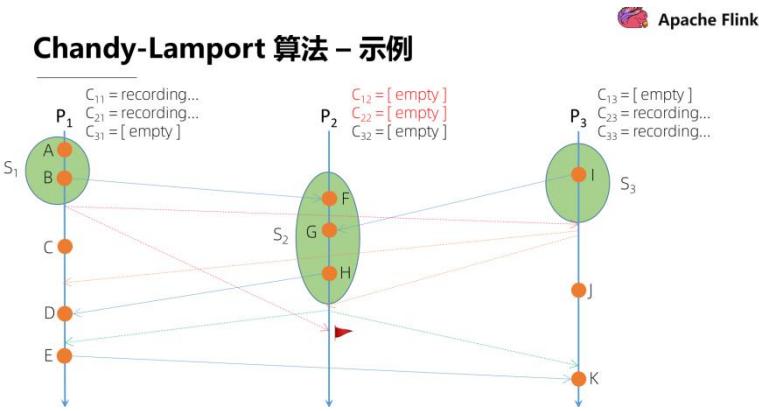
接收到 P_3 发出的 marker 信息的是 P_1 ，但这不是它接收的第一个 marker，它会把来自 C_{31} channel 的管道立刻关闭，并且把当前的记录消息做这个 channel 的快照，后续再接收到来自 P_3 的消息，就不会更新在此次的快照状态里了。



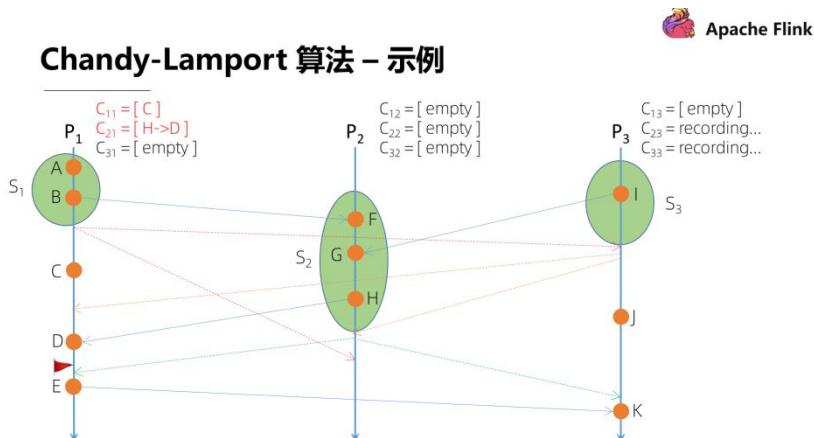
Chandy-Lamport 算法 – 示例



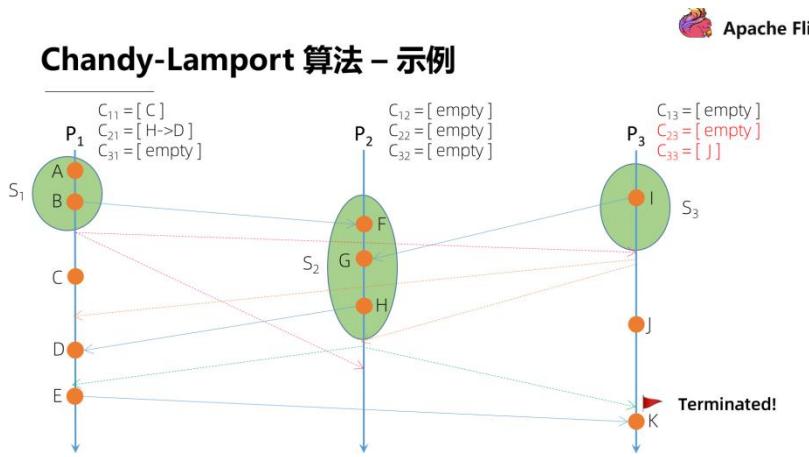
接下来 P2 接收到来自 P3 的消息，这是它接到的第一个 marker 消息。接收到消息后，它首先对本地状态进行快照，然后把 C32 管道的标记成 close，与此同时开始向它所有的 output channel 发送 marker 消息，最后它会把来自除了 C32 之外的所有 input channel 的消息开始进行记录。



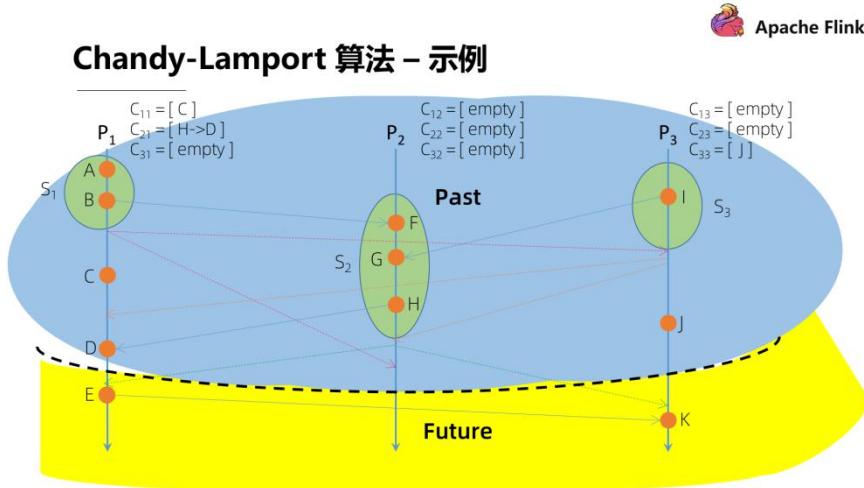
再来看 P2 接收到来自 P1 的消息，这不是 P2 接收到的第一个 marker 消息，所以它会把所有的 input channel 全部关闭，并且记录 channel 的状态。



接下来看 P1 接收到来自 P2 的消息，这也不是它接收的第一个消息。那么它就会把所有的 input channel 关闭，并把记录的消息作为状态。那么这里面有两个状态，一个是 C11，即自己发给自己的消息；一个是 C21，是 P2 里 H 发给 P1D 的。



最后一个时间点，P3 接收到来自 P2 的消息，这也不是它收到的第一个消息，操作跟上面介绍的一样。在这期间 P3 本地有一个事件 J，它也会把 J 作为它的状态。



当所有进程都记录了本地状态，而且每一个进程的所有输入管道都已经关闭了，那么全局一致性快照就结束了，也就是对过去时间点的全局性的状态记录完成了。

Chandy-Lamport 与 Flink 之间的关系

Flink 是分布式系统，所以 Flink 会采用全局一致性快照的方式形成检查点，来支持故障恢复。Flink 的异步全局一致性快照算法跟 Chandy-Lamport 算法的区别主要有以下几点：

- 第一，Chandy-Lamput 支持强连通图，而 Flink 支持弱连通图；
- 第二，Flink 采用的是裁剪的 (Tailored) Chandy-Lamput 异步快照算法；
- 第三，Flink 的异步快照算法在 DAG 场景下不需要存储 Channel state，从而极大减少快照的存储空间。

三、Flink 的容错机制



流计算容错的一致性保证

容错，即恢复到出错前的状态。这里的“出错”包含多种可能的原因，比如由于网络问题导致的 worker 失联，意外导致的进程 crash，应用程序错误等等。
对于错误恢复，Flink 可以提供不同级别的一致性保证



Exactly once (严格一次)
每条 event 会且只会对 state 产生一次影响
注意：并非端到端的严格一次



At least once (最少一次)
每条 event 会对 state 产生最少一次影响
(存在重复处理)



At most once (最多一次)
每条 event 会对 state 产生最多一次影响
注意：所有状态会在出错时丢失

Apache Flink Community China | 极客训练营

容错，就是恢复到出错前的状态。流计算容错一致性保证有三种，分别是：
Exactly once, At least once, At most once。

- Exactly once，是指每条 event 会且只会对 state 产生一次影响，这里的“一次”并非端到端的严格一次，而是指在 Flink 内部只处理一次，不包括 source 和 sink 的处理。
- At least once，是指每条 event 会对 state 产生最少一次影响，也就是存在重复处理的可能。
- At most once，是指每条 event 会对 state 产生最多一次影响，就是状态可能会在出错时丢失。

(一) 端到端的 Exactly once

Exactly once 的意思是，作业结果总是正确的，但是很可能产出多次；所以它的要求是需要有可重放的 source。

端到端的 Exactly once，是指作业结果正确且只会被产出一次，它的要求除了有可重放的 source 外，还要求有事务型的 sink 和可以接收幂等的产出结果。

(二) Flink 的状态容错

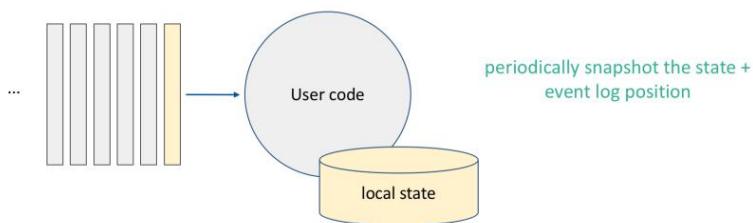
很多场景都会要求在 Exactly once 的语义，即处理且仅处理一次。如何确保语义呢？

简单场景的 Exactly Once 容错方法

简单场景的做法如下图，方法就是，记录本地状态并且把 source 的 offset，即 Event log 的位置记录下来就好了。

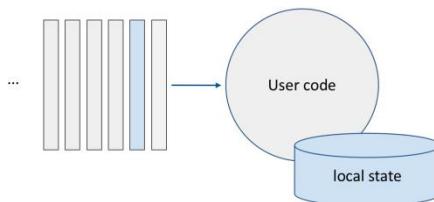


简单场景的 exactly once 容错方法

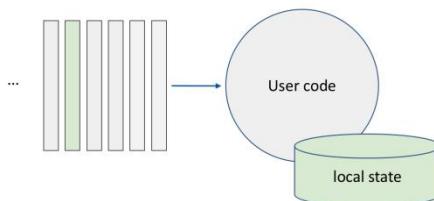




简单场景的 exactly once 容错方法



简单场景的 exactly once 容错方法



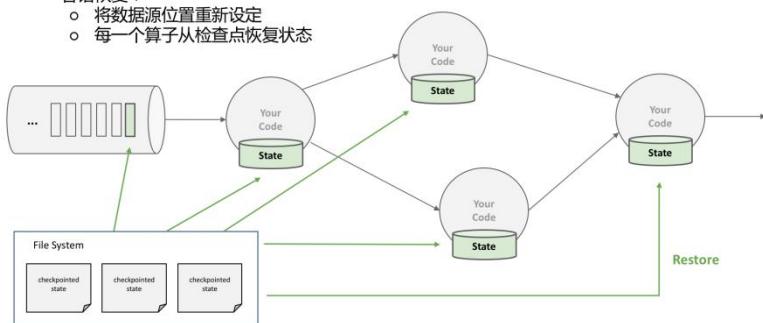
分布式场景的状态容错

如果是分布式场景，我们需要在不中断运算的前提下对多个拥有本地状态的算子产生全局一致性快照。Flink 分布式场景的作业拓扑比较特殊，它是有向无环并且是弱联通图，可以采用裁剪的 Chandy-Lamport，也就是只记录所有输入的 offset 和各个算子状态，并依赖 rewindable source（可回溯的 source，即可以通过 offset 读取比较早一点时间点），从而不需要存储 channel 的状态，这在存在聚合（aggregation）逻辑的情况下可以节省大量的存储空间。



Flink 分布式场景的状态容错

- 容错恢复：
 - 将数据源位置重新设定
 - 每一个算子从检查点恢复状态

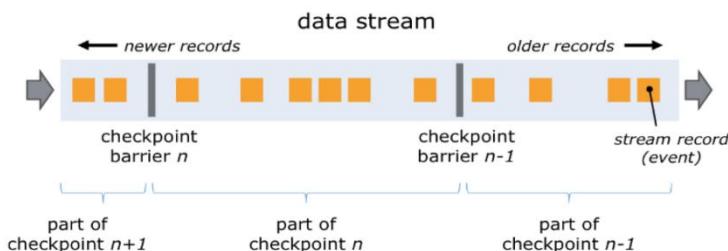


最后做恢复，恢复就是把数据源的位置重新设定，然后每一个算子都从检查点恢复状态。

(三) Flink 的分布式快照方法

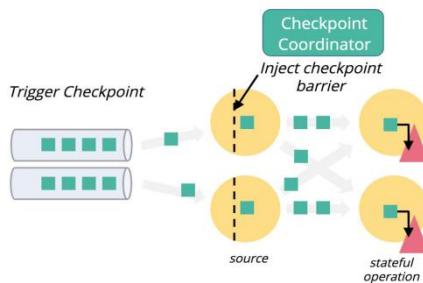


Flink 的分布式快照方法 – Checkpoint barrier



首先在源数据流里插入 Checkpoint barrier，也就是上文提到的 Chandy-Lamport 算法里的 marker message，不同的 Checkpoint barrier 会把流自然地切分多个段，每个段都包含了 Checkpoint 的数据；

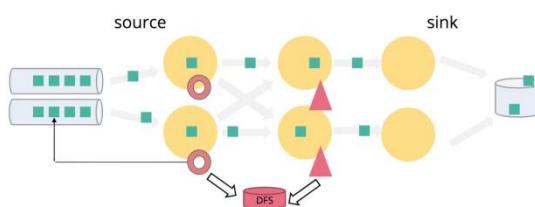
Flink 的分布式快照方法 – 启动快照



Flink 里有一个全局的 Coordinator，它不像 Chandy-Lamport 对任意一个进程都可以发起快照，这个集中式的 Coordinator 会把 Checkpoint barrier 注入到每个 source 里，然后启动快照。当每个节点收到 barrier 后，因为 Flink 里面它不存储 Channel state，所以它只需存储本地的状态就好。

Flink 的分布式快照方法 – 收到 barrier 后

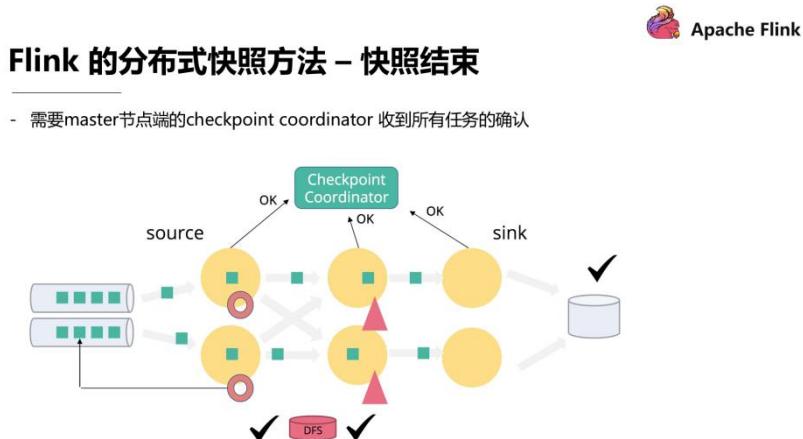
- Source 保存其输入数据的 offsets
- 算子(Operator) 保存其 state
- 事务型 Sink 节点对已有事务提交 pre-commit



在做完了 Checkpoint 后，每个算子的每个并发都会向 Coordinator 发送一个确认消息，当所有任务的确认消息都被 Checkpoint Coordinator 接收，快照就结束了。

(四) 流程演示

见下图示，假设 Checkpoint N 被注入到 source 里，这时 source 会先把它正在处理分区的 offset 记录下来。

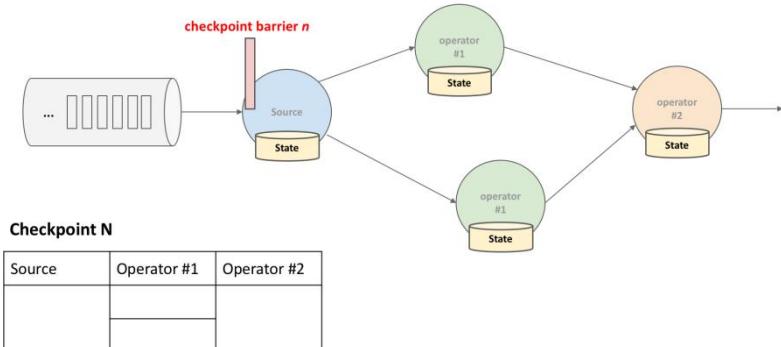


随着时间的流逝，它会把 Checkpoint barrier 发送到两个并发的下游，当 barrier 分别到达两个并发，这两个并发会分别把它们本地的状态都记录在 Checkpoint 的里。

最后 barrier 到达最终的 subtask，快照就完成了。



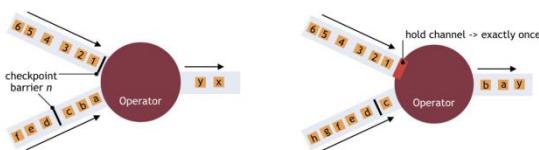
快照流程示意图



这是比较简单的场景演示，每个算子只有单流的输入，再来看下图比较复杂的场景，算子有多流输入的情况。



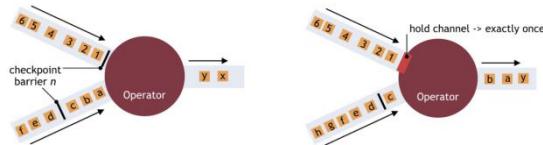
Barrier 对齐流程 – 对齐过程中



当算子有多个输入，需要把 Barrier 对齐。怎么把 Barrier 对齐呢？如下图所示，在左侧原本的状态下，当其中一条 barrier 到达，另一条 barrier 命令上有的 barrier 还在管道中没有到达，这时会在保证 Exactly once 的情况下，把先到达的流直接阻塞掉，然后等待另一条流的数据处理。等到另外一条流也到达了，会把之前的流 unblock，同时把 barrier 发送到算子。



Exactly-once vs. At-least-once



- 如果不在对齐过程中阻塞已收到 barrier 的数据管道，会发生什么？
- 本次 checkpoint 中会包含一些属于下个 checkpoint 的数据，恢复后由于 source 会 rewind，部分数据会有重复处理 -> at-least-once!!
- 如果能接受 at-least-once，那么可以选择其以避免 barrier 对齐带来的副作用

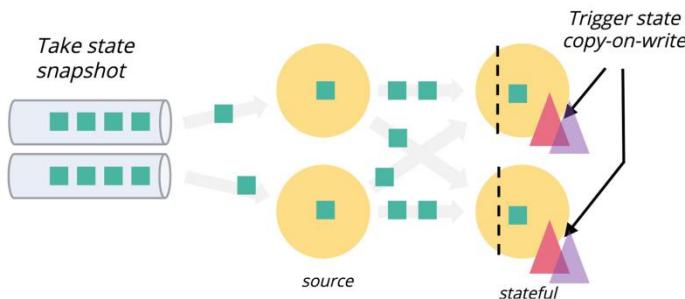
在这个过程中，阻塞掉其中一条流的作用是，会让它产生反压。Barrier 对齐会导致反压和暂停 operator 的数据处理。

如果不在对齐过程中阻塞已收到 barrier 的数据管道，数据持续不断流进来，那么属于下个 Checkpoint 的数据被包含在当前的 Checkpoint 里，如果一旦发生故障恢复后，由于 source 会被 rewind，部分数据会有重复处理，这就是 at-least-once。如果能接收 at-least-once，那么可以选择其他可以避免 barrier 对齐带来的副作用。另外也可以通过异步快照来尽量减少任务停顿并支持多个 Checkpoint 同时进行。

(五) 快照触发



快照触发 state copy-on-write

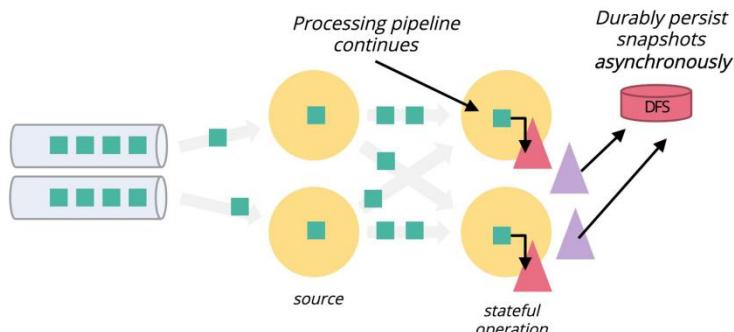


本地快照同步上传到系统需要 state Copy-on-write 的机制。

假如对元数据信息做了快照之后数据处理恢复了，在上传数据的过程中如何保证恢复的应用程序逻辑不会修改正在上传的数据呢？实际上不同状态存储后端的处理是不一样的，Heap backend 会触发数据的 copy-on-write，而对于 RocksDB backend 来说 LSM 的特性可以保证已经快照的数据不会被修改。



快照异步进行持久化



四、Flink 的状态管理

(一) Flink 状态管理



首先需要去定义一个状态，在下图的例子里，先定义一个 Value state。

在定义的状态的时候，需要给出以下的几个信息：

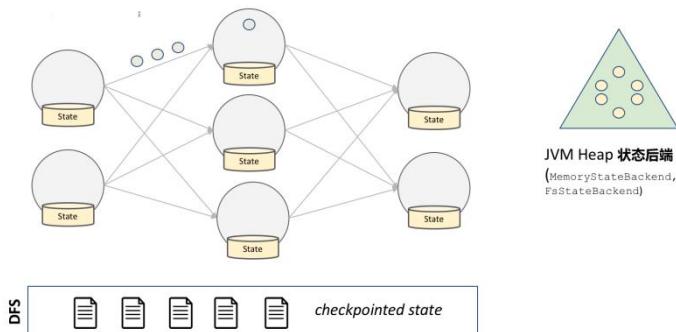
- 状态识别 ID
- 状态数据类型
- 本地状态后端注册状态
- 本地状态后端读写状态

(二) Flink 状态后端

又叫 state backend，Flink 状态后端有两种；



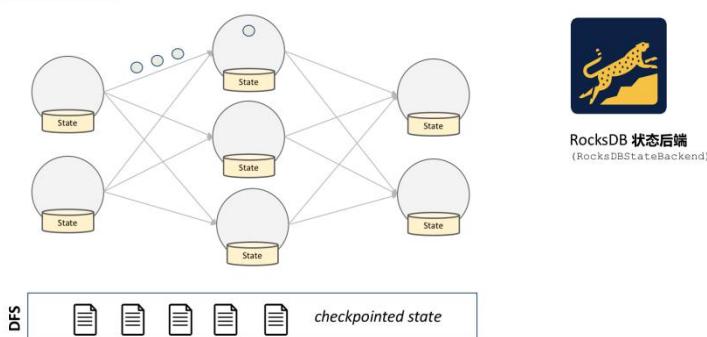
Flink 本地状态后端 – JVM Heap



第一种，JVM Heap，它里面的数据是以 Java 对象形式存在的，读写也是以对象形式去完成的，所以速度很快。但是也存在两个弊端：第一个弊端，以对象方式存储所需的空间是磁盘上序列化压缩后的数据大小的很多倍，所以占用的内存空间很大；第二个弊端，虽然读写不用做序列化，但是在形成 snapshot 时需要做序列化，所以它的异步 snapshot 过程会比较慢。



Flink 本地状态后端 – RocksDB



第二种， RocksDB，这个类型在读写时就需要做序列化，所以它读写的速度比较慢。但是它有一个好处，基于 LSM 的数据结构在快照之后会形成 sst 文件，它的异步 checkpoint 过程就是文件拷贝的过程，CPU 消耗会比较低。

Flink SQL_Table 介绍与实战

作者：伍翀（云邪）

Apache Flink PMC，阿里巴巴技术专家

本文由 Apache Flink PMC，阿里巴巴技术专家伍翀（云邪）分享，主要介绍了 Flink SQL 和 Table API 的诞生背景、概念和功能，并通过三个实例演练让观众更直观地了解了 Flink 及其在 Kibana 上的具体操作流程。内容如下：

- Flink SQL 和 Table API 诞生的背景
- Flink SQL 和 Table API 的核心概念及功能
- 结合 Demo 进行实战演练

Flink 有非常强大的 API 抽象能力，它提供了三层的 API，从底至上分别是 Process Function，DataStream API 以及 SQL 和 Table API。这三层都有不同的用户群体，越低层灵活度越高，门槛也会越高，最高层门槛较低，但是会牺牲一些灵活度。

Flink 强大的抽象能力

不同层次的抽象覆盖各类应用场景



一、为什么要花精力做 SQL 和 Table API?

DataStream API 非常好用，因为它的表达能力非常强，用户可以维护和更新应用状态，而且它对时间的控制力也非常灵活。但相对而言，它的复杂度和门槛也更高，并不适用于所有人，很多用户希望专注于业务逻辑。所以，要提供更加简单易懂的 API，SQL 是目前最佳的选择。

Flink SQL 和 Table API 优势有很多。首先它非常易于理解，很多不同行业不同领域的人都懂 SQL，它已经成为大数据处理生态圈的标准语言了；其次 SQL 是声明式的语言，用户只需要表达想要什么，而无需关心如何计算；然后 SQL 是会自动优化的，能生成最优的执行计划；同时 SQL 还是 30 多年的语言，非常稳定；最后，SQL 可以更容易地统一流和批，用同一套系统就能同时处理，让用户只关注最核心的业务逻辑。

Flink SQL / Table API 的优势



二、SQL 和 Table API 简介

Flink 关系式的 API 主要暴露两种，一种是 SQL 的 API，还有一种是 Table 的 API。SQL API 完全遵循 ANSI SQL 的标准设计，所以如果有 SQL 基础，它的学

习门槛是比较低的，而 Table 可以理解为类 SQL 的编程式的 API。他们都是统一的批处理和流处理的 API，不管输入是静态的批处理数据，还是无限的流处理数据，他的查询的结果都是相同的。总结而言，就是一份代码，一个结果，这也是流批统一的最重要的评价指标。

Apache Flink's Relational APIs

ANSI SQL

```
SELECT user, COUNT(url) AS cnt  
FROM clicks  
GROUP BY user
```

LINQ-style Table API

```
tableEnvironment  
    .scan("clicks")  
    .groupBy('user')  
    .select('user', 'url.count as 'cnt)
```

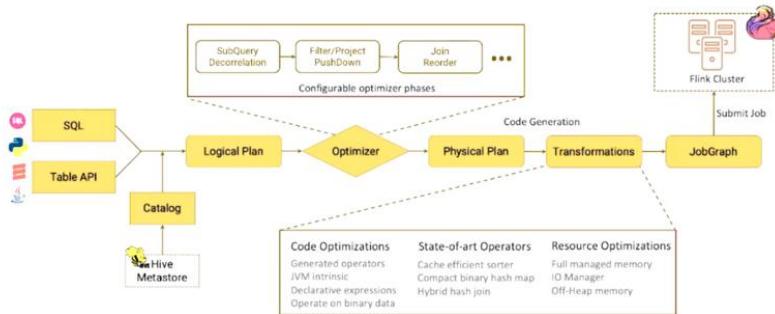
统一的 API 处理 batch & streaming 数据

一个查询描述了相同结果
不管它的输入是静态的批数据还是流数据

三、Flink 的工作流程

下面是比较高级的概览图，SQL 和 Table 在进入 Flink 以后会转化成统一的数据结构表达形式，即 Logical Plan。其中，Catalog 会提供一些原数据信息，用于后续的优化。Logical Plan 是优化的路口，经过一系列的优化规则后，Flink 会把初始的 Logical Plan 优化为 Physical Plan，并通过 Code Generation 机制翻译为 Transformation，最后转换成 JobGraph，用于提交到 Flink 的集群做分布式的执行。可以看到，整个流程并没有单独的流处理和批处理的路径，因为这些优化的过程和扩建都是共享的。

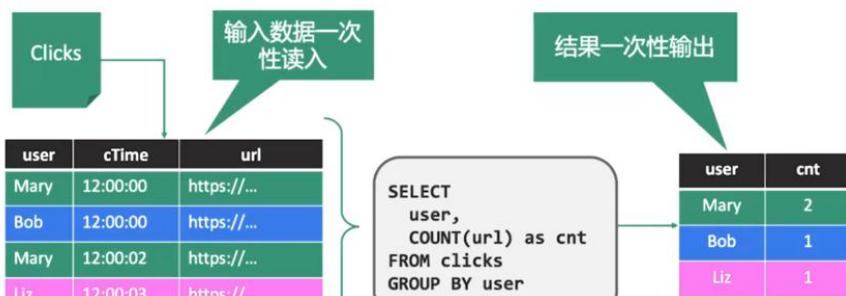
Flink SQL/Table 查询翻译



四、用实例理解流和批

比如一个点击的文件，有 user、点击的时间和 URL。如果我们要统计点击的次数，在选出 user 做统一的批处理的情况下，它的特点就是一次性读入和一次性输出。

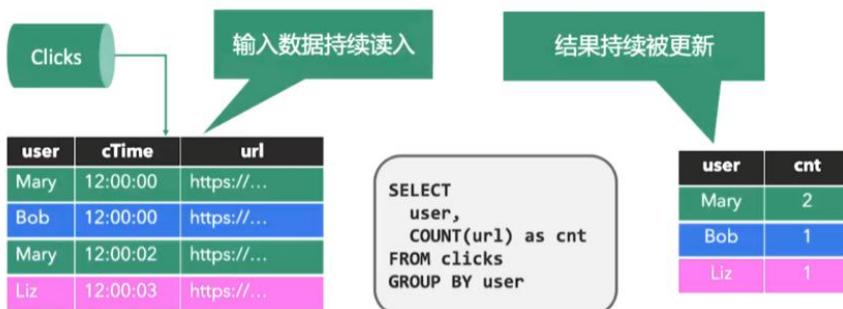
假如“Clicks”是一个文件？



而如果 Click 是一个数据流，在这种情况下，输入一条数据后就能输出一个结果，比如 Mary 第一次点击会记录一次，第二次点击就会做增量计算。所以输入数据会持续读入，结果也会持续被更新。

可以看到，这里流和批的结果是一样的，所以我们可以把以前批处理的 SQL 迁移到 Flink 上做流处理，它的结果和语义应该和之前的批处理是一样的。

假如“Clicks”是一个流？



五、Flink SQL 和 Table 应用案例

典型的包括低延迟 ETL 处理，比如数据的预处理、清洗和过滤；

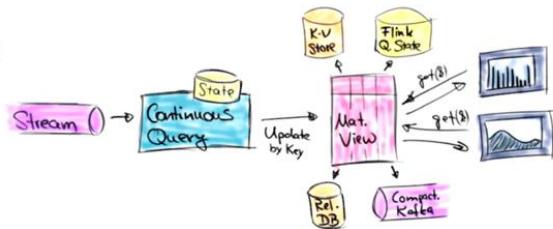
还有数据管道，Flink 可以做实时和离线的数据管道，可以构建低延时实时数仓，也可以实时数据同步，把数据从某一个数据系统同步到另一个数据系统；

第三种是流式和批式的数据分析，去计算和更新离线或实时的数据，并进行可视化，典型的比如阿里双 11 的大屏；

最后一种是模式识别，也就是实时地识别数据流中符合某种 pattern 的事件流，然后做相应的监控或者报警的服务，比如网约车的一些异常事件的监控服务。

Flink SQL/Table 应用案例

- 低延迟 ETL 处理
- 数据管道，构建低延时实时数仓，实时数据同步
- 流式 & 批式 的数据分析
 - 计算更新实时/离线数据并可视化
- 模式识别
- 网约车异常事件监测服务



六、Flink 的核心功能

下图包含了 Flink 的一些核心功能。第一个是 SQL 的 DDL，DDL 直接对接外部系统，它的强弱决定了 Flink 与外部系统的联通性，而作为一个计算引擎，与外部数据存储的联通性非常重要；第二是完整的类型系统，它支持多种数据类型，这对 SQL 引擎而言也是非常必要的；第三是高效流式 TopN，有非常强大的流处理能力，用来实时计算排行榜，比如双 11 的销量排行榜；还有高效的流式去重对数据进行过滤，因为有时采集会包含重复的数据；还有维表关联、对接 CDC 等。

除此之外，Flink 还有非常多的内置函数，支持 MiniBatch，以及有多种解热点手段。它还支持完整的批处理，适用 Python 等语言，还有 Hive 的集成等功能，不仅能直接访问 Hive 的数据，还兼容了 Hive 的语法，让用户不必再频繁切换。

Flink SQL/Table 核心功能一览 (v1.11)



示例

下面是一个电商的用户行为的实时分析。我们从 Kafka 中实时地消费用户的行
数据，并与 MySQL 中的数据进行关联，然后写入 Elasticsearch 的索引中，并用
Kibana 进行视觉化呈现。这是一个端到端的实时应用的构建。

电商用户行为实时分析

Apache Flink



下面是在 Kibana 上的最终展示成果，会有面板进行实时监控，显示出包括当前
的独立用户数、类目排行、各时段购买量等数据。



下面是来自某宝的用户行为日志，我们只选取了 11 月 27 日当天的行为，它包含这几个字段，包括用户 ID、商品 ID、商品类目 ID、行为类型和时间戳。行为类型中，pv 代表点击，buy 代表购买，cart 代表加入购物车，fav 代表收藏事件，而时间戳代表事件发生的时间。

用户行为日志



列名称	说明
用户ID	整数类型，序列化后的用户ID
商品ID	整数类型，序列化后的商品ID
商品类别ID	整数类型，序列化后的商品所属类别ID
行为类型	字符串，枚举类型，包括{'pv', 'buy', 'cart', 'fav'}
时间戳	行为发生的时间戳

2017-11-27 当天的行为：数据生成器每秒生成2000条数据

数据来源：阿里云天池公开数据集

七、实战演练

演练的示例代码已经传到了 Github，大家如果有兴趣也可以按照这个文档一步一步做下去。我们准备一台装有 Docker 的 Linux 或者 MacOS 计算机即可，不需要下载额外的包。

首先，我们新建一个目录，比如叫 flink-sql-demo，然后把 docker-compose 的 demo 文件下载下来，可以点进去看一下这个文件。



```

准备
一台带有 Docker 的 Linux 或 MacOS 计算机。
使用 Docker Compose 启动容器
本实践将从 GitHub 上获取的组件重新部署到了容器中，因此可以通过 docker-compose 一键启动。也可以通过 wget 命令或直接编辑 docker-compose.yml 文件，也可以手动下载。
$ cd flink-sql-demo; cd flink-sql-demo;
$ wget https://raw.githubusercontent.com/wuchengf/Flink-SQL-Demo/v5.13/docker-compose.yml
通过 Docker Compose 安装的容器有
• Flink SQL Client: 用于提交 Flink SQL。
• Flink 集群: 包含一个 JobManager 和一个 TaskManager 用于运行 SQL 任务。
• DataGen: 生成代理容器，帮助我们自动向 MySQL 或者 Kafka 中插入数据，并发送到 Kafka 集群中。默认每秒生成 2000 条数据，耗时大概生成一两个小时，也可以更改 docker-compose.yml 中 datagen 的 numtasks 参数来调整生成速率（重启 docker compose 才能生效）。
• MySQL: 构造了 MySQL 5.7，以及预先部署好了商品表（category），预先插入了大量的与商品相关的映射关系，后续作为训练数据。
• Kafka: 主要用来存储数据，DataGen 后端会自动将数据插入这个消费者中。
• Zookeeper: Kafka 的数据依赖。
• Elasticsearch: 主要存储 Flink SQL 产生的数据。
• Kibana: 可视化 Elasticsearch 产生的数据。
在启动而集群，建议修改 Docker 的配置，将资源调整到 4GB 以及 4 核。启动所有的容器，只需要运行 docker-compose.yml 并在普通环境下如下命令。

```

这里面有个 datagen 的数据源，我们可以去控制它的产生速度，比如把产生的速度从 2000 改成 3000。

```

users@wuchong:~/Downloads/flink-sql-demo$ docker-compose up -d --no-deps flink-sql-demo
[...]

```

我们通过 docker-compose up-d 把 docker 中的容器都启动起来。容器包括 Jobmanager、Taskmanager 这两个 Flink 的集群，还有 Kibana、Elasticsearch、Zookeeper、MySQL、Kafka 等。

```

正在连接 raw.githubusercontent.com (raw.githubusercontent.com) 10.0.0.0:443... 失败: Connection refused,
正在连接 raw.githubusercontent.com (raw.githubusercontent.com) 10.0.0.0:443... 失败: Connection refused,
+ Flink-sql-demo wget https://raw.githubusercontent.com/wuchong/flink-sql-demo/v1.11-CN/docker-compose.yml
--2020-07-19 18:02:14- https://raw.githubusercontent.com/wuchong/flink-sql-demo/v1.11-CN/docker-compose.yml
正在解析主机 raw.githubusercontent.com (raw.githubusercontent.com)... ::, 0.0.0.0
正在连接 raw.githubusercontent.com (raw.githubusercontent.com):443... 失败: Connection refused,
正在连接 raw.githubusercontent.com (raw.githubusercontent.com):443... 失败: Connection refused,
+ Flink-sql-demo wget https://raw.githubusercontent.com/wuchong/flink-sql-demo/v1.11-CN/docker-compose.yml
--2020-07-19 18:02:20- https://raw.githubusercontent.com/wuchong/flink-sql-demo/v1.11-CN/docker-compose.yml
正在解析主机 raw.githubusercontent.com (raw.githubusercontent.com)... ::, 0.0.0.0
正在连接 raw.githubusercontent.com (raw.githubusercontent.com):443... 失败: Connection refused,
正在连接 raw.githubusercontent.com (raw.githubusercontent.com):443... 失败: Connection refused,
+ Flink-sql-demo ls
docker-compose.yml
+ Flink-sql-demo code docker-compose.yml
+ Flink-sql-demo docker-compose up -d
Creating network "flink-sql-demo_default" with the default driver.
Creating Flink-sql-demo_jobmanager_1 ... done
Creating Flink-sql-demo_kibana_1 ... done
Creating Flink-sql-demo_zookeeper_1 ... done
Creating Flink-sql-demo_elasticsearch_1 ... done
Creating Flink-sql-demo_mysql_1 ... done
Creating Flink-sql-demo_taskmanager_1 ... done
Creating Flink-sql-demo_kafka_1 ... done
Creating Flink-sql-demo_sql-client_1 ... done
Creating Flink-sql-demo_datagen_1 ... done
+ Flink-sql-demo docker-compose exec kafka bash -c 'kafka-console-consumer.sh --topic user_behavior --bootstrap-server
kafka:9094 --from-beginning --max-messages 10'

```

我们可以用 Docker-compose 的命令看一下 Kafka 中最新的 10 条数据。它有 user ID，有商品 ID，有类目 ID，有用户的行为，还有一个 TS 代表这个行为当时发生的时间。

```
正在连接 now.githubusercontent.com (now.githubusercontent.com):0.0.0.0:443... 失败: Connection refused.
+ flink-sql-demo ls
docker-compose.yml
+ flink-sql-demo code docker-compose.yml
+ flink-sql-demo docker-compose up -d
Creating network "flink-sql_demo_default" with the default driver
Creating flink-sql_demo_jobmanager_1 ... done
Creating flink-sql_demo_kibana_1 ... done
Creating flink-sql_demo_zookeeper_1 ... done
Creating flink-sql_demo_elasticsearch_1 ... done
Creating flink-sql_demo_mysql_1 ... done
Creating flink-sql_demo_taskmanager_1 ... done
Creating flink-sql_demo_kafka_1 ... done
Creating flink-sql_demo_sql-client_1 ... done
Creating flink-sql_demo_datagen_1 ... done
+ flink-sql_demo docker-compose exec kafka bash < kafka-console-consumer.sh --topic user_behavior --bootstrap-server
kafka:9094 --from-beginning --max-messages 10
("user_id": "952483", "item_id": "310884", "category_id": "4580532", "behavior": "pv", "ts": "2017-11-27 00:00:00")
("user_id": "794777", "item_id": "5119439", "category_id": "982926", "behavior": "pv", "ts": "2017-11-27 00:00:00")
("user_id": "875914", "item_id": "4484065", "category_id": "1324293", "behavior": "pv", "ts": "2017-11-27 00:00:00")
("user_id": "940877", "item_id": "5097986", "category_id": "149192", "behavior": "pv", "ts": "2017-11-27 00:00:00")
("user_id": "944074", "item_id": "2948782", "category_id": "5001561", "behavior": "pv", "ts": "2017-11-27 00:00:00")
("user_id": "973127", "item_id": "1132597", "category_id": "4181361", "behavior": "pv", "ts": "2017-11-27 00:00:00")
("user_id": "84681", "item_id": "3505100", "category_id": "2465336", "behavior": "pv", "ts": "2017-11-27 00:00:00")
("user_id": "732136", "item_id": "3815446", "category_id": "2342116", "behavior": "pv", "ts": "2017-11-27 00:00:00")
("user_id": "940143", "item_id": "2157435", "category_id": "1013319", "behavior": "pv", "ts": "2017-11-27 00:00:00")
("user_id": "655789", "item_id": "4945338", "category_id": "4145813", "behavior": "pv", "ts": "2017-11-27 00:00:00")
Processed a total of 10 messages
+ flink-sql-demo
```

随后我们启动今天的主角，通过 Docker-compose 启动 SQL-Client 容器，当看到这个松鼠的时候，SQL Client 就成功启动了，我们可以在里面运行 SQL 的命令。



第一步我们要用 DDL 创建数据源，把用户日志的数据源先创建起来。我们用 Create Table 这个 DDL 语法创建了一个 user behavior 的表，它里面有 5 个字段，包括 user ID，商品 ID，类目 ID，用户行为和 TS 时间戳。With 里面跟的是一些如何连接到外部系统的属性，比如用 Kafka 连接外部的 topic。

```

CREATE TABLE user_behavior
    WITH (
        'connector' = 'kafka',
        'topic' = 'user_behavior',
        'group.id' = 'user_behavior_group',
        'format' = 'json',
        'scan.startup.mode' = 'earliest-offset',
        'properties.bootstrap.servers' = 'kafka:9094',
        'format' = 'json'
    )
    PROCTIME AS PROCTIME(), -- generates processing-time attribute using computed column
    WATERMARK FOR ts AS ts - INTERVAL '5' SECOND -- defines watermark on ts column, marks ts as event-time attribute
    ;

    user_id BIGINT,
    item_id BIGINT,
    category_id BIGINT,
    behavior STRING,
    ts TIMESTAMP(3),
    proctime AS PROCTIME(), -- generates processing-time attribute using computed column
    WATERMARK FOR ts AS ts - INTERVAL '5' SECOND -- defines watermark on ts column, marks ts as event-time attribute
) WITH (
    'connector' = 'kafka', -- using kafka connector
    'topic' = 'user_behavior', -- kafka topic
    'scan.startup.mode' = 'earliest-offset', -- reading from the beginning
    'properties.bootstrap.servers' = 'kafka:9094', -- kafka broker address
    'format' = 'json' -- the data format is json
);

有了数据源后，我们就可以用 SQL 去读取并连接这个 Kafka 中的 topic 了，看 Flink SQL CLU 中执行语句。

```

在上面的截图里我们列出了 5 个字段，除此之外，我们还通过计算函数和 WATERMARK 内置函数声明了一个产生事件时间的属性。我们还通过 WATERMARK 语句，在 ts 字段上声明了 watermark 策略（标记水印策略），当接收的数据晚于事件时间时，等于时间属性以后的 ts，通过它可以知道是否发生了解水印。

• 参阅文档：<https://ci.apache.org/projects/flink/flink-docs-release-1.17/dev/table/streaming-time-attributes.html>

• DOCS：<https://ci.apache.org/projects/flink/flink-docs-release-1.17/table/table.html#createFromKafkaTable>

在 SQL CLU 中成功创建 Kafka 表后，可以通过 show tables 和 describe user_behavior 来查看目前创建的

另外我们也能通过 show table 看 user behavior，用 describe table 看表的结构、字段、计算列、watermark 策略等等。

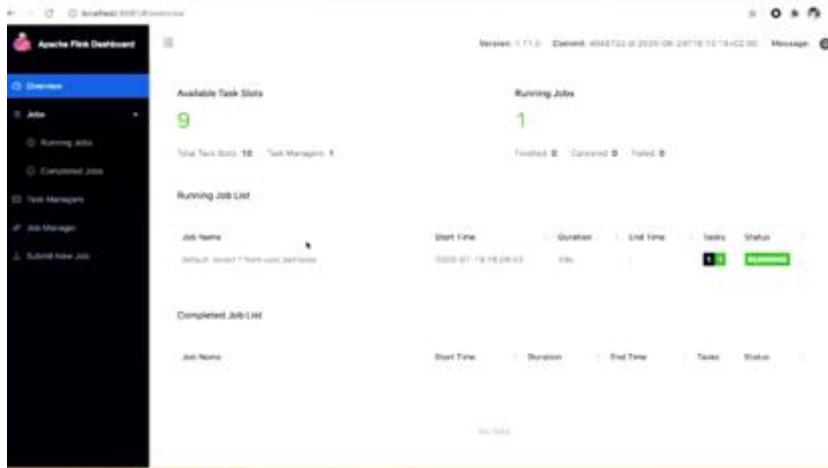
```

link: \sql> show tables;
user_behavior

link: \sql> describe user_behavior;
+-----+
| user_id          | BIGINT          |
| item_id          | BIGINT          |
| category_id      | BIGINT          |
| behavior         | STRING          |
| ts               | TIMESTAMP(3)   |
| proctime         | PROCTIME       |
| WATERMARK FOR ts | ts - INTERVAL '5' SECOND
+-----+

```

我们也可以进到 8081 端口，这是 Docker compose 旗下 Flink 集群的一个 Web UI 界面，这里面各个栏目大家都可以去看看。



接下来我们用 3 个实战去画一些图表，深入了解 Flink 的一些功能。

首先是统计每小时的成交量。我们先用 DDL 创建 Elasticsearch 表，定义每小时的成交量，随后提交 Query 去做每小时成交量的统计分析。

我们需要做每小时的一个滑窗，用到 Tumble Window 语法。Tumble 的第一个字段定义时间属性，也就是刚刚说的 TS 事件时间，然后开窗大小是一小时，也就是说我们每小时会滑一个窗口，然后对窗口内的数据做统计分析。

```

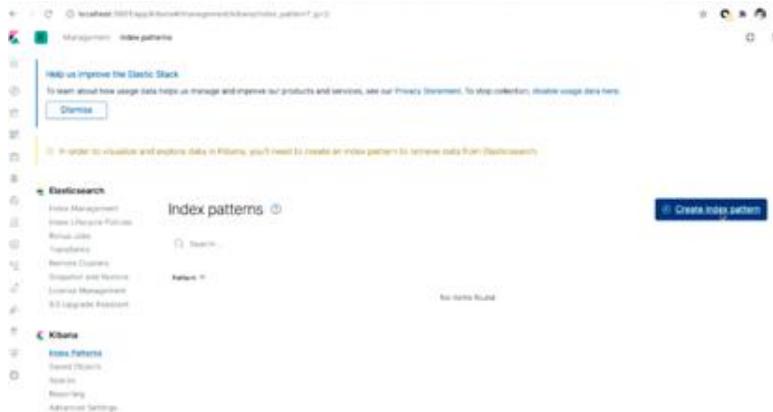
    * http://localhost:8081/applications/flink-test-stm
      累计每小时的成交量

    使用 DDL 创建 Elasticsearch 表
    我们先在 SQL 中创建一个 ES 的表名，根据对数据需求主要将要保存两个数据：小时、成交量。
    CREATE TABLE log_hourly AS
    SELECT hour, sum AS total
    FROM log
    GROUP BY hour
    INTO TABLE log_hourly
    WITH (
        'connection' = 'elasticsearch://127.0.0.1:9200',
        'index' = 'log_hourly',
        'type' = 'log_hour'
    );

    我们不需要在 Elasticsearch 中手动创建 log_hourly 索引，Flink SQL 会自动创建索引。
    提交 Query
    但以每个小时的成交量就是每个小时有多少 "key" 的用户行为。因此需要使用到 FlinkSQL 窗口函数。按照一小时切窗，然后每个窗口内统计所有 "key" 的个数。这可以通过先汇总 "key" 的数据，然后 COUNT+1 实现。
    SELECT COUNT(key) AS key_count, hour
    FROM log_hourly
    GROUP BY hour
    AND key
    ORDER BY key ASC, hour DESC LIMIT 10000;
  
```

注意我们使用 hour 内置函数，这是一个 TIMESTAMP 的字段指出一小时中第几个小时的数据。仅限于 3.6.0 和 3.7.0 版本。query 不会直接从新版本的 Flink 变更文件中移除（可以在 [as 旧参数被解释 query 的简化规则](#)）。有关此主题，请参阅 [了解更多关于窗口聚合的内容](#) <https://cwiki.apache.org/confluence/display/Flink/Flink+Data+Release-1.11#DataRelease-1.11IndexPatternManagement>

我们提交这个 Query，然后通过 5601 端口访问 Kibana 对它进行可视化。刚进来的时候是空的，里面什么数据都没有，所以我们一般要先创建 create index pattern，通过页面的 Management 中的 Index Pattern 进入，找到我们的索引，点击进去创建。



创建了 Index Pattern 以后，我们才能在当中做一些 Discovery 或者可视化。

可以看到，这些字段就是我们刚才 DDL 里定义的，并且它还有对应的值。



当然我们最终是要进行可视化，所以我们要创建一个 Dashboard。在页面左上角点击 Dashboard，然后点击 Create New 创建新的视图，随后就可以设置每小时的成交量了。



我们画一个面积图，在 Y 轴上选择购买量 max，标签名字改为“成交量”，然后因为 X 轴展示时间，所以选择“hour-of-day”，order by 字母序，改为 24，随后点击播放键，面积图就画好了。我们也可以点击保存，这样这个面积图就会保存到 Dashboard 上。



随后我们再画一个图，统计一天每 10 分钟累计独立用户数。同样，我们需要现在 SQL CLI 中创建一个 Elasticsearch 表，用于存储结果汇总数据，字符段包括日期时间和累计 uv 数。



统计一天每10分钟累计独立用户数

我们有相应的同构化语句对一天中每一秒的累计独立用户数 (uv)，也就是说每一秒的 uv 数据应该是从 0 到到当前秒数为 1 的累加 uv 值，因此语句肯定是要递归调用的。

我们在原先 SQL CLI 中创建一个 Elasticsearch 表，用于存储成天汇总数据。主要字段有：日期时间戳和累积 uv 数，以及 UV 在时间点为 Elasticsearch 中的 document_id，便于查询该日期时间的 uv 值。

```
CREATE TABLE cumulative_uv
    date_str STRING,
    time_str STRING,
    uv BIGINT,
    id BIGINT
    WITH (
        'connector' = 'elasticsearch-7',
        'hosts' = "http://elasticsearch:9200",
        'index' = "cumulative_uv"
    );
```

为了方便清晰，我们可以再通过 CLOCK WINDOW 计算出每天的数据的当前分钟，以及日期戳 (or TaskTime) 开始到现在的时分为上的窗口 (分钟)。uv 语句我们通过内置的 COUNT(DISTINCT user_id) 来完成。Flink SQL 内置的 COUNT DISTINCT 做了单步的优化，因此可以放心使用。

为了方便清晰，我们可以先在日志文件中运行，我们使用 `time_window` 拿取出基本的日期和时间，再用 `mapWith` 和 `mapToPair` 来构建结果集：将时间戳 (或日期) 和分钟值 (如 12:10, 12:20, 等等) 作为 key，将独立用户的数量 (或日数) 作为 value。来构造最大的时间 (或 UV) 可以到 Elasticsearch 的索引中，UV 的统计我们通过内置的 COUNT(DISTINCT user_id) 来实现。

然后，我们在 SQL CLI 中执行 Table。

```
> WITH (
    'connector' = 'elasticsearch-7', -- using elasticsearch connector
    'hosts' = "http://elasticsearch:9200", -- elasticsearch address
    'index' = 'buy_cnt_per_hour' -- elasticsearch index name, similar to database table name
);
[INFO] Table has been created.

link SQL> INSERT INTO buy_cnt_per_hour
SELECT HOUR(CLOCK_WINDOW(ts, INTERVAL '1' HOUR)), COUNT(*)
FROM user_behavior
WHERE behavior = 'buy';
GROUP BY TUMBLE(ts, INTERVAL '1' HOUR);
[INFO] Submitting SQL update statement to the cluster...
[INFO] Table update statement has been successfully submitted to the cluster;
id: 2498e17923cd8c04bd216856d89937d

|
link SQL> CREATE TABLE cumulative_uv (
    date_str STRING,
    time_str STRING,
    uv BIGINT,
    PRIMARY KEY (date_str, time_str) NOT ENFORCED
) WITH (
    'connector' = 'elasticsearch-7',
    'hosts' = "http://elasticsearch:9200",
    'index' = 'cumulative_uv'
);
[INFO] Table has been created.
```

这里面 Query 主要做了一件事，就是把日期和时间选出来。这里唯一有一点特殊的是，因为我们的需求是做每 10 分钟的点，所以用 Substr 的两个竖线做连接的函数来实现。随后，我们像之前一样把 Query 提交到 SQL CLI 里面运行。

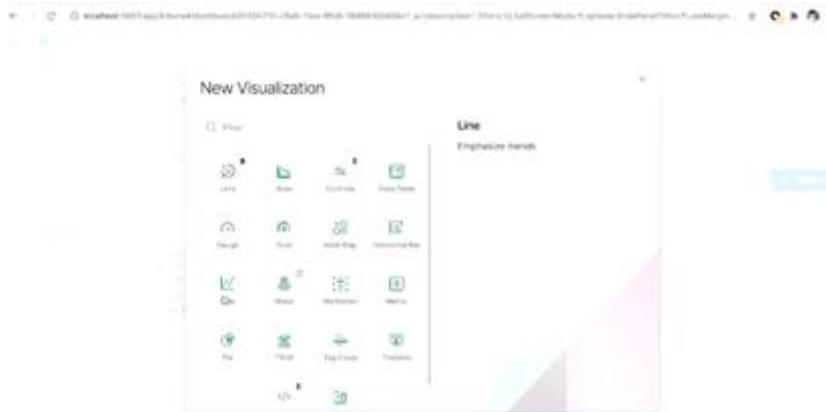


```

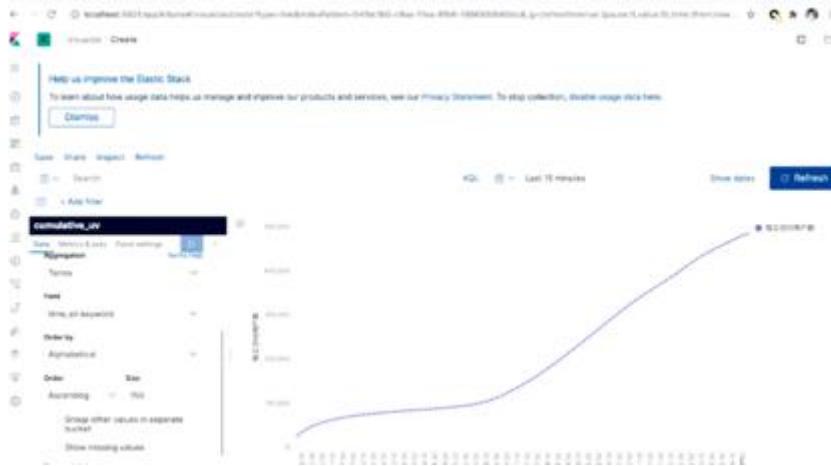
gulshan@Gulshans-MacBook-Pro:~/Desktop$ ./bin/sql-client.sh -c "SELECT date_str, COUNT(DISTINCT user_id) AS uv FROM uv GROUP BY date_str"
+-----+-----+
| date_str | uv |
+-----+-----+
| 2018-01-01 | 1000 |
| 2018-01-02 | 1000 |
| 2018-01-03 | 1000 |
| 2018-01-04 | 1000 |
| 2018-01-05 | 1000 |
| 2018-01-06 | 1000 |
| 2018-01-07 | 1000 |
| 2018-01-08 | 1000 |
| 2018-01-09 | 1000 |
| 2018-01-10 | 1000 |
| 2018-01-11 | 1000 |
| 2018-01-12 | 1000 |
| 2018-01-13 | 1000 |
| 2018-01-14 | 1000 |
| 2018-01-15 | 1000 |
| 2018-01-16 | 1000 |
| 2018-01-17 | 1000 |
| 2018-01-18 | 1000 |
| 2018-01-19 | 1000 |
| 2018-01-20 | 1000 |
| 2018-01-21 | 1000 |
| 2018-01-22 | 1000 |
| 2018-01-23 | 1000 |
| 2018-01-24 | 1000 |
| 2018-01-25 | 1000 |
| 2018-01-26 | 1000 |
| 2018-01-27 | 1000 |
| 2018-01-28 | 1000 |
| 2018-01-29 | 1000 |
| 2018-01-30 | 1000 |
| 2018-01-31 | 1000 |
+-----+-----+

```

和之前一样，我们创建新的视图，这里我们创建的是一个连线图。



我们在 Y 轴取 uv 的值，命名为“独立访问用户数”，X 轴选 terms，然后选 time-str，order by Alphabetical，一天中改为 150 个点。然后点播放，独立用户数的曲线图就出现了。同样，我们可以点击保存，把这个图加到 Dashboard 上。



随后我们来画第三张图。第三张图是顶级类目排行榜，因为一个商品对应的类目太细分，比如对应非常细的第三四级类目，所以它对排行榜意义可能不大。但是我们希望归约到一个顶级类目做统计分析，所以开始之前准备了 MySQL 的容器，里面准备了子类目和顶级类目的映射关系。

我们首先在 SQL CLI 中创建 MySQL 表，后续用作维表查询，同时再创建一个 Elasticsearch 表，用于存储类目统计结果。而在 Query 这里，我们会用到 Create View 语法去注册一个临时的视图，简化写法，因为把两个 Query 写在一起可能会比较复杂。

```
④ http://www.elasticsearch.org/guide/en/elasticsearch/guide/1.4/_index.html
```

顶级类目排行榜

最后一个有意思的可视化是类目排行榜，从而了解哪些类目最受欢迎。不过由于该数据中的类目实在太多（约5000个类目），对于用户来说意义不大。因此我们希望将所有的类目分成几大类，所以笔者在 MySQL 中首先准备了子类目与一级类目的映射关系，具体如图。

在 SQL 中创建 MySQL 表，并使用作表语句。

```
CREATE TABLE category_dim (
    category_id INT,
    category_name STRING,
    category_level INT
);
CREATE TABLE category_fact (
    category_id INT,
    category_name STRING,
    category_level INT,
    count INT
);

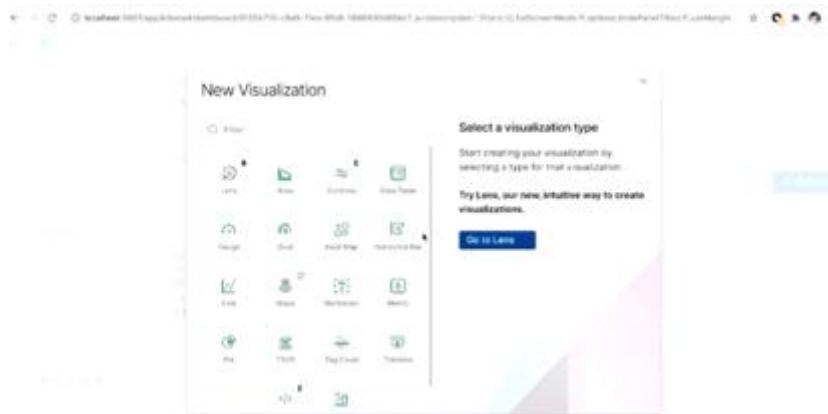
INSERT INTO category_dim
SELECT category_id, category_name, category_level
FROM category;

INSERT INTO category_fact
SELECT category_id, category_name, category_level, COUNT(*) AS count
FROM category
GROUP BY category_id, category_name, category_level;
```

同时我们再搭建一个 Elasticsearch 索引，用于存储类目计数。

```
PUT /category/_index
{
    "mappings": {
        "category": {
            "properties": {
                "category_id": {"type": "integer", "index": "not_analyzed"},  
                "category_name": {"type": "string", "index": "not_analyzed"},  
                "category_level": {"type": "integer", "index": "not_analyzed"},  
                "count": {"type": "integer", "index": "not_analyzed"}  
            }
        }
    }
}
```

我们同样在 SQL CLI 中运行代码，然后进入到 Kibana 页面建立索引和添加可视化图表。这里我们使用 Horizontal Bar 去画一个柱形图。



在 Y 轴上，我们统计类目的成交量，X 轴用类目的名字，排序用倒序排列，随后点击播放键。



最后，我们同样点击保存，把类目排行榜添加到 Dashboard 上。加上之前我们做的两个，Dashboard 上就有 3 个图表了，这里可以自己拖拽图表美化一下。



以上就是今天的课程，大家也可以到 Github 上的文档中再去学习和实践。

PyFlink 快速上手

作者：付典

Apache Flink PMC，阿里巴巴技术专家

本文介绍了 PyFlink 项目的目标和发展历程，以及 PyFlink 目前的核心功能，包括 Python Table API、Python UDF、向量化 Python UDF、Python UDF Metrics、PyFlink 依赖管理和 Python UDF 执行优化，同时也针对功能展示了相关 demo。本文主要分为 4 个部分：

- PyFlink 介绍
- PyFlink 相关功能
- PyFlink 功能演示
- PyFlink 下一步规划

一、PyFlink 介绍

PyFlink 是 Flink 的一个子模块，也是整个 Flink 项目的一部分，主要目的是提供 Flink 的 Python 语言支持。因为在机器学习和数据分析等领域，Python 语言非常重要，甚至是最主要的开发语言。所以，为了满足更多用户需求，拓宽 Flink 的生态，我们启动了 PyFlink 项目。

PyFlink项目的目标



PyFlink 项目的目标主要有两点，第一点是将 Flink 的计算能力输出给 Python 用户，也就是我们会在 Flink 中提供一系列的 Python API，方便对 Python 语言比较熟悉的用户开发 Flink 作业。

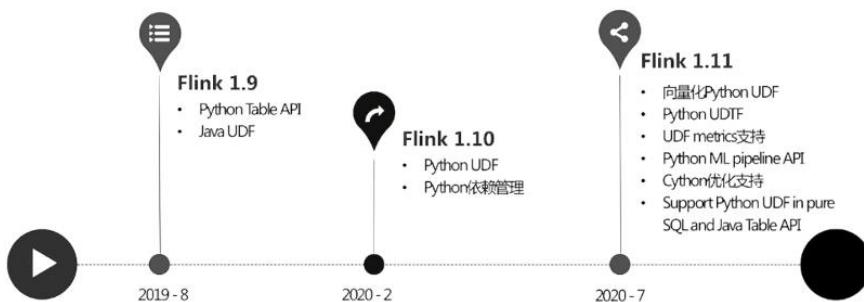
第二点，就是将 Python 生态基于 Flink 进行分布式化。虽然我们会在 Flink 中提供一系列的 Python API 来给 Python 用户来使用，但这对用户来说是有学习成本的，因为用户要学习怎么使用 Flink 的 Python API，了解每一个 API 的用途。所以我们希望用户能在 API 层使用他们比较熟悉的 Python 库的 API，但是底层的计算引擎使用 Flink，从而降低他们的学习成本。这是我们未来要做的事情，目前处于启动阶段。

PyFlink项目的目标



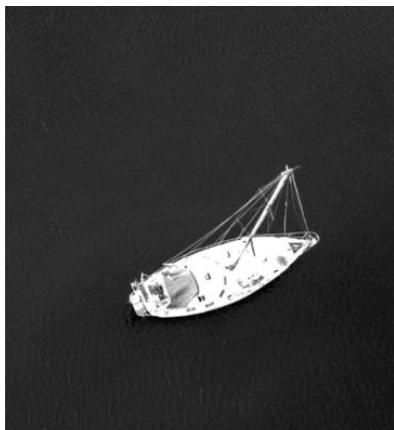
下图是 PyFlink 项目的发展情况，目前发布了 3 个版本，支持的内容也越来越丰富。

PyFlink项目的发展历程



二、PyFlink 相关功能介绍

我们主要介绍 PyFlink 以下功能，Python Table API、Python UDF、向量化 Python UDF、Python UDF Metrics、PyFlink 依赖管理和 Python UDF 执行优化。



PyFlink功能介绍

- Python Table API
- Python UDF
- 向量化Python UDF
- Python UDF Metrics
- PyFlink依赖管理
- Python UDF执行优化

(一) Python Table API

Python Table API 的目的是为了让用户可以使用 Python 语言来开发 Flink 作业。Flink 里面有三种类型的 API，Process、Function 和 Table API，前两者是较为底层的 API，基于 Process 和 Function 开发的作业，其逻辑会严格按照用户定义的行为进行执行，而 Table API 是较为高层的 API，基于 Table API 开发的作业，其逻辑会经过一系列的优化之后进行执行。

Python Table API，顾名思义就是提供 Table API 的 Python 语言支持。



Python Table API

目标

- 支持用户使用Python语言编写Flink作业
- Python Table API功能丰富，能完成Java Table API所支持的绝大部分功能

以下是 Python Table API 开发的一个 Flink 作业，作业逻辑是读取文件，计算 word count，然后再把计算结果写到文件中去。这个例子虽然简单，但包括了开发一个 Python Table API 作业的所有基本流程。

首先我们需要定义作业的执行模式，比如说是批模式还是流模式，作业的并发度是多少？作业的配置是什么。接下来我们需要定义 source 表和 sink 表，source 表定义了作业的数据源来源于哪里，数据的格式是什么；sink 表定义了作业的执行结果写到哪里去，数据格式是什么。最后我们需要定义作业的执行逻辑，在这个例子中是计算写过来的 count。



Apache Flink

Python Table API - 示例

```

from pyflink.table import EnvironmentSettings, BatchTableEnvironment
# Init environment
environment_settings = EnvironmentSettings.newBuilder().use_blink_planner().in_batch_mode().build()
t_env = BatchTableEnvironment.create(environment_settings)
t_env.set_parallelism(1)

# Register Source
t_env.execute_sql("""
    CREATE TABLE mySource (
        word STRING
    ) WITH (
        'connector' = 'filesystem',
        'format' = 'csv',
        'path' = '/opt/examples/data/word_count_input'
    )
""")

# Register Sink
t_env.execute_sql("""
    CREATE TABLE mySink (
        word STRING,
        `count` BIGINT
    ) WITH (
        'connector' = 'filesystem',
        'format' = 'csv',
        'path' = '/opt/examples/data/word_count_output'
    )
""")

# Query
t_env.sql_query("SELECT word, COUNT(*) AS `count` FROM mySource") \
    .group_by("word") \
    .select("word, count(*)") \
    .insert_into("mySink")

# Execute
t_env.execute("1-word count")

```

1. 初始化环境

2. 定义一个source表

3. 定义一个sink表

4. 定义作业逻辑

Apache Flink Community China | 极客训练营

以下是 Python Table API 的部分截图，可以看到它的数量和功能都比较齐全。



(二) Python UDF

Python Table API 是一种关系型的 API，其功能可以类比成 SQL，而 SQL 里自定义函数是非常重要的功能，可以极大地扩展 SQL 的使用范围。Python UDF 的主要目的就是允许用户使用 Python 语言来开发自定义函数，从而扩展 Python Table API 的使用场景。同时，Python UDF 除了可以用在 Python Table API 作业中之外，还可以用在 Java Table API 作业以及 SQL 作业中。

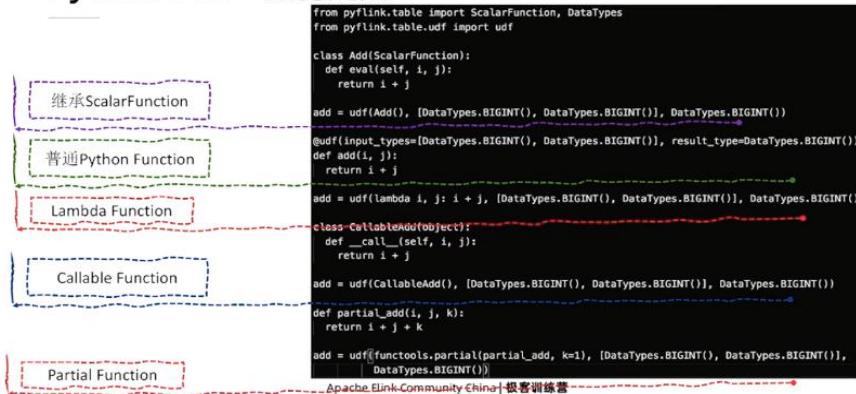
Python UDF

目标

- 支持用户自定义和使用Python UDF
 - Python UDF除了可以使用在Python Table API作业中外，还可以使用在Java Table API作业以及SQL作业中

在 PyFlink 中我们支持多种方式来定义 Python UDF。用户可以定义一个 Python 类，继承 ScalarFunction，也可以定义一个普通的 Python 函数或者 Lambda 函数，实现自定义函数的逻辑。除此之外，我们还支持通过 Callable Function 和 Partial Function 定义 Python UDF。用户可以根据自己的需要选择最适合自己的方式。

Python UDF - 如何定义



PyFlink 里面提供了多种 Python UDF 的使用方式，包括 Python Table API、Java table API 和 SQL，我们一一介绍。

在 Python Table API 中使用 Python UDF，在定义完 Python UDF 之后，用户首先需要注册 Python UDF，可以调用 table environment register 来注册，然后命名，然后就可以在作业中通过这个名字来使用 Python UDF 了。



Python UDF - 如何使用

- Python Table API作业中使用Python UDF

```
table_env.register_function("function_name", python_udf)
```

```
35 # Register the Python function
36 udf = udf(lambda i: i + 1, DataTypes.BIGINT(), DataTypes.BIGINT())
37 table_env.register_function("add_one", udf)
38
39 # Use the Python function in Python Table API
40 my_table.select("a, add_one(b)")
```

NOTE: Python Table API作业中也可以使用Java UDF

```
table_env.register_java_function("function_name", "java.userdefined.function.class.name")
```

Apache Flink Community China | 极客训练营

在 Java Table API 中它的使用方式也比较相似，但是注册方式不一样，Java Table API 作业中需要通过 DDL 语句来进行注册。



Python UDF - 如何使用

- Java Table API作业中使用Python UDF

```
public class BlinkBatchPythonUdfSqlJob {
    public static void main(String[] args) {
        TableEnvironment tEnv = TableEnvironment.create(
            EnvironmentSettings.newInstance().useBlinkPlanner().inBatchMode().build());
        tEnv.getConfig().getConfiguration().setCoreOptions.DEFAULT_PARALLELISM, 2);
        tEnv.getConfig().getConfiguration().setPythonOptions.USE_AWAKE_REPORT, true);
        tEnv.executeSql("create temporary system function add_one as 'udfs.add_one' language python");

        tEnv.createTemporaryView("source", tEnv.fromValues(1L, 2L, 3L).as("a"));

        Iterator<Row> result = tEnv.executeSql("select add_one(a) as from source").collect();
        List<Long> actual = new ArrayList<Long>();
        while(result.hasNext()) {
            Row r = result.next();
            actual.add((Long) r.getField(0));
        }

        List<Long> expected = Arrays.asList(2L, 3L, 4L);
        if (!actual.equals(expected)) {
            throw new AssertionError(String.format("The output result: %s is not as expected: %s!", actual, expected));
        }
    }
}
```

1. 注册Python UDF

2. 在Java作业中使用Python UDF

作业提交: flink run -j pyflink-playgrounds.jar -c BlinkBatchPythonUdfSqlJob -pyfs /opt/examples/utils/udfs.py

Apache Flink Community China | 极客训练营

除此之外，用户也可以在 SQL 的作业中使用 Python UDF。与前面两种方式类似，用户首先需要注册 Python UDF，可以在 SQL 脚本中通过 DDL 语句来注册，也可以在 SQL Client 的环境配置文件里面注册。

Python UDF - 如何使用



- SQL作业中使用Python UDF

- 注册Python UDF

sql-client.yaml

方式1: create temporary system function add_one as 'udfs.add_one' language python;

方式2: SQL client环境配置文件

```

table:
  sink:
    type: sink-table
    update-mode: append
    schema:
      - name: a
        type: BIGINT
    connector:
      type: filesystem
      path: "/opt/examples/data/sql-test-out/result.csv"
    format:
      type: csv
      fields:
        - name:
          type: BIGINT

functions:
  - name: add_one
    from: python
    fully-qualified-name: udfs.add_one
    configuration:
      python_fn-execution.memory.managed: true

```

Apache Flink Community China | 极客训练营

→ 注册Python UDF

Python UDF 架构

简单介绍下 Python UDF 的执行架构。Flink 是用 Java 语言编写的，运行在 Java 虚拟机中，而 Python UDF 运行在 Python 虚拟机中，所以 Java 进程和 Python 进程需要进行数据通信。除此之外，两者间还需要传输 state、log、metrics，它们的传输协议需要支持 4 种类型。



Python UDF - 如何使用

- SQL作业中使用Python UDF

sql-client.yaml

方式1: create temporary system function add_one as 'udfs.add_one' language python;

方式2: SQL client环境配置文件

```

table:
  - name: sink
    type: sink-table
    update: true
    append: true
    schema:
      - name: a
        type: BIGINT
    connector:
      type: filesystem
      path: "/opt/examples/data/sql-test-out/result.csv"
    format:
      type: csv
      fields:
        - name: a
          type: BIGINT
functions:
  - name: add_one
    from: python
    fully-qualified-name: udfs.add_one
configuration:
  python.fn-execution.memory.managed: true

```

Apache Flink Community China | 极客训练营

注册Python UDF

(三) 向量化 Python UDF

向量化 Python UDF 的主要目的是使 Python 用户可以利用 Pandas 或者 Numpy 等数据分析领域常用的 Python 库，开发高性能的 Python UDF。



向量化Python UDF

目标

- 支持在Flink Java/Python Table API & SQL作业中自定义和使用向量化Python UDF
- 方便Python用户基于Pandas、Numpy等数据分析领域常用的Python库，开发高性能的Python UDF

向量化 Python UDF 是相对于普通 Python UDF 而言的，我们可以在下图看到两者区别的区别。



向量化Python UDF

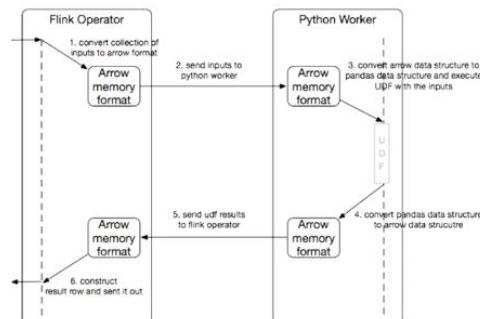
普通Python UDF VS 向量化Python UDF

	普通Python UDF	向量化Python UDF
数据处理方式	每次UDF调用，处理一条数据	每次UDF调用，处理N条数据
序列化 / 反序列化	针对每条数据，在Java端和Python端都需要序列化 / 反序列化	1) 由于Pandas原生支持Apache Arrow，Python端无需序列化 / 反序列化 2) Java端，有机会做向量化优化，同时可以尽量避免序列化 / 反序列化
执行性能	较差	较好： 1) 向量化执行，效率高 2) Pandas / NumPy等类库性能高、可以基于Pandas / NumPy等类库实现高性能Python UDF

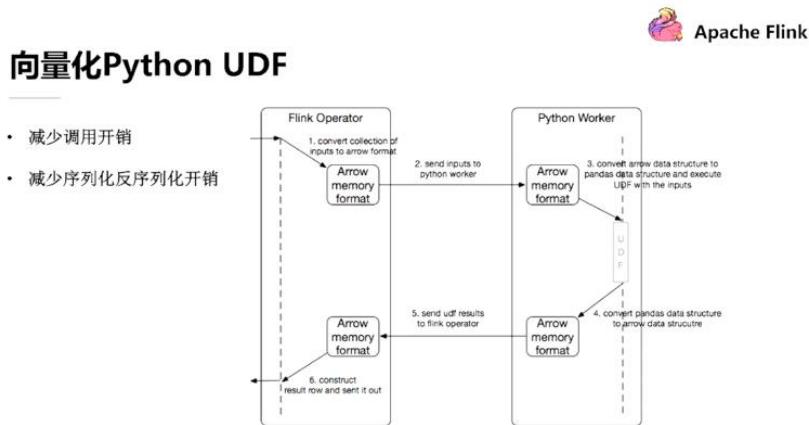
下图显示了向量化 Python UDF 的执行过程。首先在 Java 端，Java 在攒完多条数据之后会转换成 Arrow 格式，然后发送给 Python 进程。Python 进程在收到数据之后，将其转换成 Pandas 的数据结构，然后调用用户自定义的向量化 Python UDF。同时向量化 Python UDF 的执行结果会再转化成 Arrow 格式的数据，再发送给 Java 进程。

向量化Python UDF

- 减少调用开销
- 减少序列化反序列化开销



在使用方式上，向量化 Python UDF 与普通 Python UDF 是类似的，只有以下几个地方稍有不同。首先向量化 Python UDF 的声明方式需要加一个 UDF type，声明这是一个向量化 Python UDF，同时 UDF 的输入输出类型是 Pandas Series。



(四) Python UDF Metrics

前面我们提到 Python UDF 有多种定义方式，但是如果需要在 Python UDF 中使用 Metrics，那么 Python UDF 必须继承 ScalarFunction 来进行定义。在 Python UDF 的 open 方法里面提供了一个 Function Context 参数，用户可以通过 Function Context 参数来注册 Metrics，然后就可以通过注册的 Metrics 对象来汇报了。



Python UDF Metrics

```

class MyUDF(ScalarFunction):
    def __init__(self):
        self.counter = None

    def open(self, function_context):
        super().open(function_context)
        self.counter = function_context.get_metric_group().counter("my_counter")

    def eval(self, i):
        self.counter.inc(3)
        return i - 1

```

—————> 注册 Metrics
—————> 汇报 Metrics

关于其他类型的Metrics的用法: <https://ci.apache.org/projects/flink/flink-docs-master/dev/table/python/metrics.html>

(五) PyFlink 依赖管理

从类型来说，PyFlink 依赖主要包括以下几种类型，普通的 PyFlink 文件、存档文件，第三方的库、PyFlink 解释器，或者 Java 的 Jar 包等等。从解决方案来看，针对每种类型的依赖，PyFlink 提供了两种解决方案，一种是 API 的解决方案，一种是命令行选项的方式，大家选择其一即可。

PyFlink 依赖管理

类型	API	命令行选项
普通Python文件	table_env.add_python_file(file_path)	--pyFiles/-pyfs
存档文件	table_env.add_python_archive("py_env.zip", "myenv") # the files contained in the archive file can be accessed in UDF def my_udf(): with open("myenv/py_env/data/data.txt") as f: ...	--pyArchives/-pyarch
三方库	# commands executed in shell echo numpy>=1.16.5 > requirements.txt pip download -d cached_dir -r requirements.txt --no-binary :all: # python code table_env.set_python_requirements("requirements.txt", "cached_dir")	--pyRequirements/-pyreq
指定Python解释器路径	table_env.get_config().set_python_executable("py_env.zip/py_env/bin/python")	--pyExecutable/-pyexec
Jar包	table_env.get_config().set_configuration("pipeline_jars", "file:///my/jar/path/connector.jar,file:///my/jar/path/udf.jar")	--jarfile/-j

(六) Python UDF 执行优化

Python UDF 的执行优化主要包括两个方面，执行计划优化和运行时优化。它与 SQL 非常像，一个包含 Python UDF 的作业，首先会经过预先定义的规则，生成一个最优的执行计划。在执行计划已经确定的情况下，在实际执行的时候，又可以运用一些其他的优化手段来达到尽可能高的执行效率。

Python UDF 执行优化



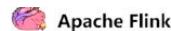
执行计划优化



运行时优化

Python UDF 执行计划优化

执行计划的优化主要有以下几个优化思路。一个是不同类型的 UDF 的拆分，由于在一个节点中可能同时包含多种类型的 UDF，而不同的类型的 UDF 是不能放在一块执行的；第二个方面是 Filter 下推，其主要目的是尽可能降低含有 Python UDF 节点的输入数据量，从而提升整个作业的执行性能；第三个优化思路是 Python UDF Chaining，Java 进程与 Python 进程之间的通信开销以及序列化反序列化开销比较大，而 Python UDF Chaining 可以尽量减少 Java 进程和 Python 进程之间的通信开销。



Python UDF执行优化 - 执行计划优化



执行计划优化



不同类型UDF的拆分

Python UDF / Vectorized Python UDF / Java UDF拆分



Filter下推到Python UDF之前

尽量减少Python UDF节点的输入数据



Python UDF Chaining

尽量减少Java/Python进程之间的数据传输

不同类型 UDF 的拆分

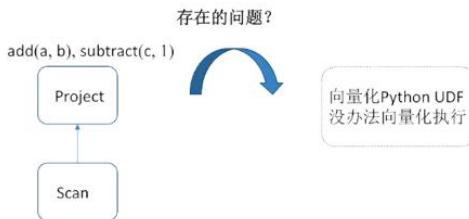
假如有这样一个作业，它包含了两个 UDF, 其中 add 是 Python UDF, subtract 是向量化 Python UDF。默认情况下，这个作业的执行计划会有一个 project 节点，这两个 UDF 同时位于这一 project 的节点里面。这个执行计划的主要问题是，普通 Python UDF 每次处理一条数据，而向量化 Python UDF，每次处理多条数据，所以这样的一个执行计划是没有办法执行的。



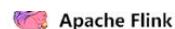
Python UDF执行优化 - 执行计划优化

- 不同类型UDF的拆分

```
table.select("add(a, b), subtract(c, 1)");
其中add普通Python UDF, subtract为
向量化Python UDF
```

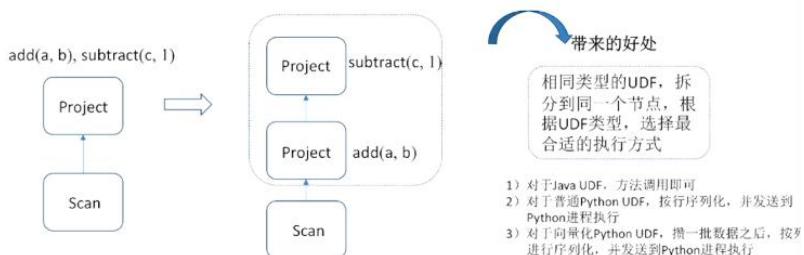


但是通过拆分，我们可以把这个一个 project 的节点拆分成了两个 project 的节点，其中第一个 project 的节点只包含普通 Python UDF，而第二个节点只包含向量化 Python UDF。不同类型的 Python UDF 拆分到不同的节点之后，每一个节点都只包含了一种类型的 UDF，所以算子就可以根据它所包含的 UDF 的类型选择最合适的执行方式。



Python UDF执行优化 - 执行计划优化

- 不同类型UDF的拆分



Filter 下推到 Python UDF 之前

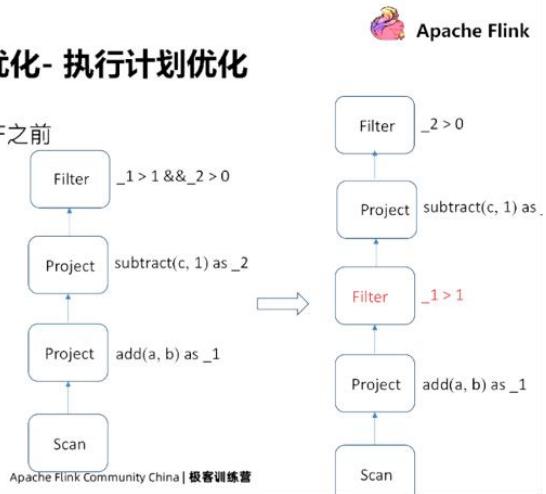
Filter 下推的主要目的是将过滤算子下推到 Python UDF 节点之前，尽量减少 Python UDF 节点的数据量。

假如我们有这样一个作业，作业原始执行计划里面包括了两个 Project 的节点，一个是 `add`、`subtract`，同时还包括一个 Filter 节点。这个执行计划是可以运行的，但需要更优化。可以看到，因为 Python 的节点位于 Filter 节点之前，所以在 Filter 节点之前 Python UDF 已经计算完了，但是如果把 Filter 过滤掉，推到 Python UDF 之前，那么就可以大大降低 Python UDF 节点的输入数据量。

Python UDF执行优化- 执行计划优化

- Filter下推到Python UDF之前

```
table.where("add(a, b) > 1 && subtract(c, 1) > 0").select("a, b, c");
其中 add 为 Java UDF， subtract 为 Python UDF
```



Python UDF Chaining

假如我们有这样一个作业，里面包含两种类型的 UDF，一个是 add，一个是 subtract，它们都是普通的 Python UDF。在一个执行计划里面包含两个 project 的节点，其中第一个 project 的节点先算 subtract，然后再传输给第二个 project 节点进行执行。

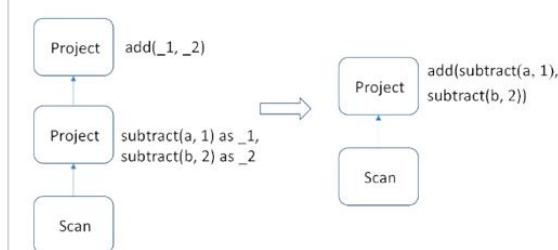
它的主要问题是，由于 subtract 和 add 位于两个不同的节点，其计算结果需要从 Python 发送回 Java，然后再由 Java 进程发送给第二个节点的 Python 进行执行。相当于数据在 Java 进程和 Python 进程之间转了一圈，所以它带来了完全不必要的通信开销和序列化反序列化开销。因此，我们可以将执行计划优化成右图，就是将 add 节点和 subtract 节点放在一个节点中运行，subtract 节点的结果计算出来之后直接去调用 add 节点。



Python UDF执行优化- 执行计划优化

- Python UDF chaining

`table.select("add(subtract(a, 1), subtract(b, 2))") ;`
其中add和subtract都是Python UDF



Python UDF 运行时优化

目前提高 Python UDF 运营时的执行效率有三种：一是 Cython 优化，用它来提高 Python 代码的执行效率；二是自定义 Java 进程和 Python 进程之间的序列化器和反序列化器，提高序列化和反序列化效率；三是提供向量化 Python UDF 功能。

Python UDF执行优化 - 运行时优化



运行时优化



Cython support



自定义序列化器



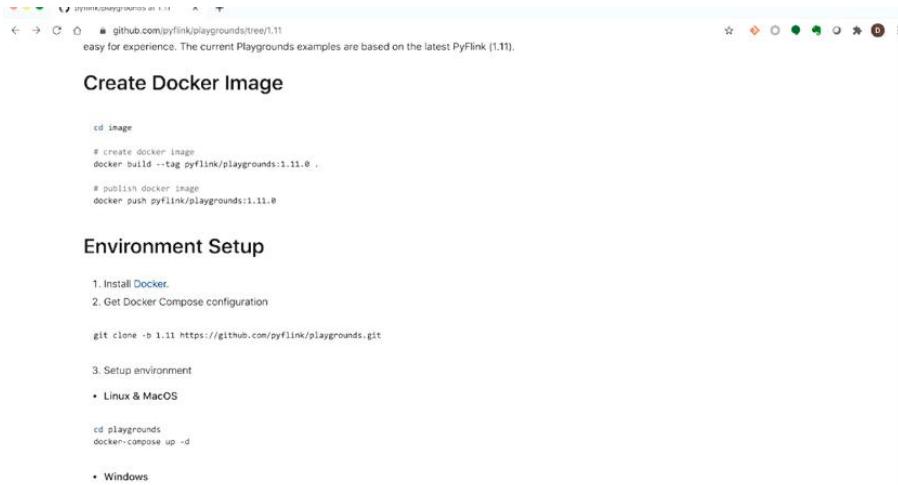
向量化Python UDF support

三、PyFlink 相关功能演示

首先大家打开这个页面，里面提供了 PyFlink 的一些 demo，这些 demo 是运行在 docker 里面的，所以大家如果要运行这些 demo 就需要在本机安装 docker 环境。


PyFlink功能演示

<https://github.com/pyflink/playgrounds/tree/1.11>



```

Create Docker Image

cd image
# create docker image
docker build --tag pyflink/playgrounds:1.11.0 .
# publish docker image
docker push pyflink/playgrounds:1.11.0

Environment Setup

1. Install Docker.
2. Get Docker Compose configuration

git clone -b 1.11 https://github.com/pyflink/playgrounds.git

3. Setup environment
• Linux & MacOS
cd playgrounds
docker-compose up -d

• Windows

```

随后，我们可以运行命令，命令会启动一个 PyFlink 的集群，后面我们运行的 PyFlink 的例子都会提交到集群去执行。

```

ali-186590@bef2d:playgrounds dianfu$ docker-compose up -d
Creating network "playgrounds_default" with the default driver
Creating playgrounds_jobmanager_1 ... done
Creating playgrounds_zookeeper_1 ... done
Creating playgrounds_kafka_1 ... done
Creating playgrounds_taskmanager_1 ... done
ali-186590@bef2d:playgrounds dianfu$ docker ps
CONTAINER ID ENVIRONMENT NAMES COMMAND CREATED STATUS PORTS
0800987f07c7 pyflink/playgrounds:1.11.0 "/docker-entrypoint..." 17 seconds ago Up 16 seconds 6121-6123/tcp, 8081/tcp
6d61307b0255 wurstmeister/kafka "start-kafka.sh" 17 seconds ago Up 16 seconds 0.0.0.0:32771->9092/tcp
56fe2dce0212 wurstmeister/zookeeper "/bin/sh -c '/usr/sbin... 19 seconds ago Up 17 seconds 22/tcp, 2888/tcp, 3888/tcp
0.0.0.0:2181->2181/tcp playgrounds_zookeeper_1
719e53da4418 pyflink/playgrounds:1.11.0 "/docker-entrypoint..." 19 seconds ago Up 17 seconds 6123/tcp, 0.0.0.0:8081->80
1/tcp 1. Get Envoy playgrounds_jobmanager_1
ali-186590@bef2d:playgrounds dianfu$ 
+ Linux & MacOS
+ Windows
+ cd playgrounds
+ docker-compose up -d
You can check whether the environment is running correctly by visiting Flink Web UI http://localhost:8081.
Google Chrome

```

第一个例子是 word count，我们首先在里面定义了环境、source、sink 等，我们可以运行一下这个作业。

```

# INIT environment
environment_settings = EnvironmentSettings.new_instance().use_blink_planner().in_batch_mode().build()
t_env = StreamTableEnvironment.create(environment_settings.environment_settings())
t_env.set_parallelism(1)

# Register Source
t_env.execute_sql("""
CREATE TABLE mySource (
    word STRING
) WITH (
    connector = 'filesystem',
    format = 'csv',
    path = '/opt/examples/data/word_count_input'
)
""")

# Register Sink
t_env.execute_sql("""
CREATE TABLE mySink (
    word STRING,
    count BIGINT
) WITH (
    connector = 'filesystem',
    format = 'csv',
    path = '/opt/examples/data/word_count_output'
)
""")

# Query
t_env.from_path('mySource') \
    .group_by('word') \
    .select('word, count()') \
    .insert_into('mySink')

# Execute
t_env.execute("3-word_count")

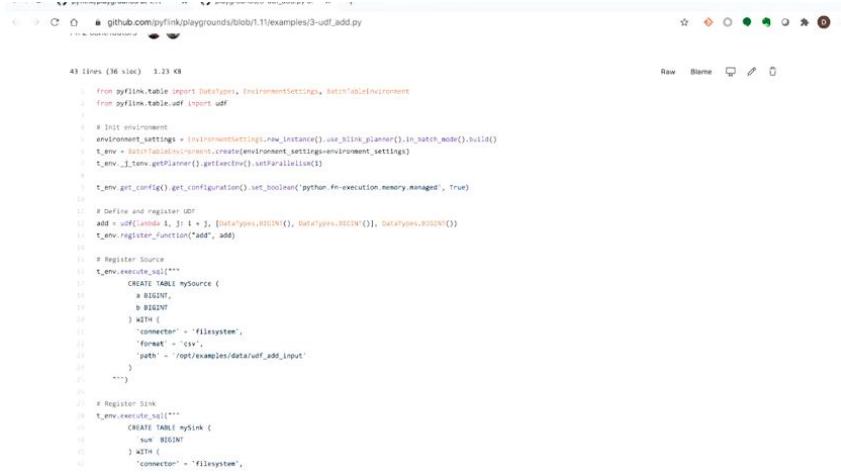
```

这是作业的执行结果，可以看到 Flink 这个单词出现了两次，PyFlink 这个单词出现了一次。

```
oli-186590cbef2d:playgrounds dianfu$ docker-compose up -d
Creating network "playgrounds_default" with the default driver
Creating playgrounds_jobmanager_1 ... done
Creating playgrounds_zookeeper_1 ... done
Creating playgrounds_kafka_1 ... done
Creating playgrounds_taskmanager_1 ... done
oli-186590cbef2d:playgrounds dianfu$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS
0888987f07c7        pyflink/playgrounds:1.11.0   "/docker-entrypoint..."   17 seconds ago    Up 16 seconds      6121-6123/tcp, 8081/tcp
6d61307b0255        wurstmeister/kafka      "start-kafka.sh"       17 seconds ago    Up 16 seconds      0.0.0.0:32771->9092/tcp
56fe2dc0e0212       wurstmeister/zookeeper   "/bin/sh -c '/usr/sbin/service kafka start'"  19 seconds ago    Up 17 seconds      22/tcp, 2888/tcp, 3888/tcp
0.0.0.0:2181->2181/tcp
playgrounds_zookeeper_1
719e93d04418        pyflink/playgrounds:1.11.0   "/docker-entrypoint..."   19 seconds ago    Up 17 seconds      6123/tcp, 0.0.0.0:8081->80
1/tcp
playgrounds_jobmanager_1
oli-186590cbef2d:playgrounds dianfu$ docker-compose exec jobmanager ./bin/flink run -py /opt/examples/1-word_count.py
Job has been submitted with JobID b35c94bdf9786680f0cd03a1c438daed
Program execution finished
Job with JobID b35c94bdf9786680f0cd03a1c438daed has finished.
Job Runtime: 2826 ms

oli-186590cbef2d:playgrounds dianfu$ cat examples/data/word_count_output/part-e748cc3f-75e2-4a1f-8c82-0b2097536a67-cp-0-task-0-file-0
flink,2
pyflink,1
oli-186590cbef2d:playgrounds dianfu$
```

接下来再运行一个 Python UDF 的例子。这个例子和前面有一些类似，首先我们定义它使用 PyFlink，运行在批这种模式下，同时作业的并发度是 1。不一样的地方是我们在作业里定义了一个 UDF，它的输入包括两个列，都是 Bigint 类型，而且它输出类型也是对应的。这个 UDF 的逻辑是把这两个列的相加作为一个结果输出。



```

33 lines (36 sloc) 3.23 KB
from pyflink.table import DataTypes, EnvironmentSettings, StreamTableEnvironment
from pyflink.table.udf import udf
...
# Init environment
environment_settings = EnvironmentSettings().use_blink_planner().in_batch_mode().build()
t_env = StreamTableEnvironment.create(environment_settings=environment_settings)
t_env._j_tEnv.getPlanner().getEnvEnv().setParallelism(1)

t_env.get_config().get_configuration().set_boolean("python.fn-execution.memory.managed", True)

# Define and register UDF
add = udf(lambda i, j: i + j, [DataTypes.BIGINT(), DataTypes.BIGINT()], DataTypes.BIGINT())
t_env.register_function("add", add)

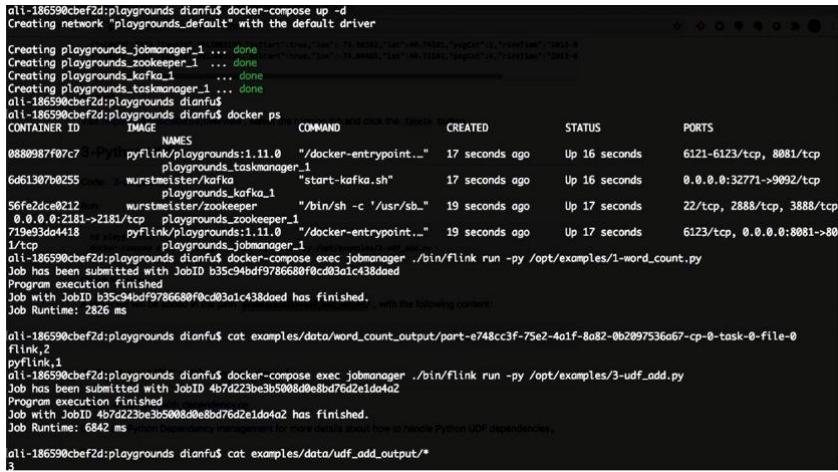
# Register Source
t_env.execute_sql("""
CREATE TABLE mysource (
    a BIGINT,
    b BIGINT
) WITH (
    'connector' = 'filesystem',
    'format' = 'csv',
    'path' = '/opt/examples/data/udf_add_input'
)
""")

# Register Sink
t_env.execute_sql("""
CREATE TABLE mysink (
    sum BIGINT
) WITH (
    'connector' = 'filesystem',
)
""")

# Register Function
t_env.register_function("add", add)

```

我们执行一下作业，执行结果是 3。



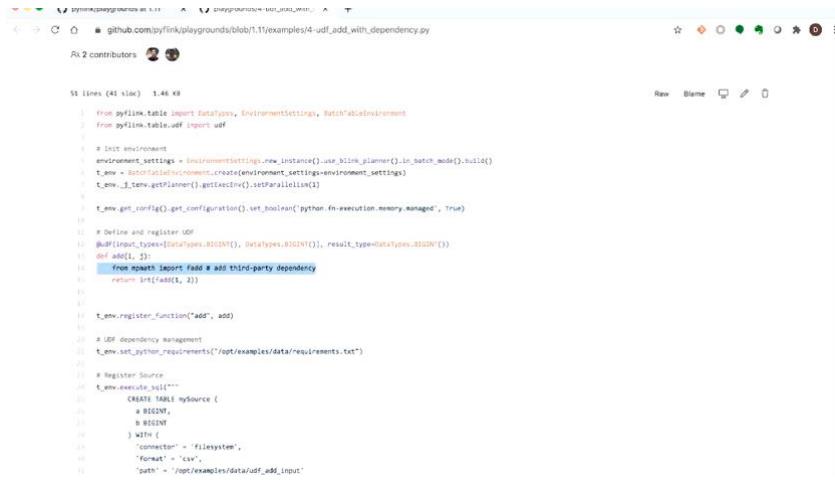
```

ali-186590@bef2d:playgrounds dianfu$ docker-compose up -d
Creating network "playgrounds_default" with the default driver
Creating playgrounds_jobmanager_1 ... done
Creating playgrounds_zookeeper_1 ... done
Creating playgrounds_kafka_1 ... done
Creating playgrounds_taskmanager_1 ... done
ali-186590@bef2d:playgrounds dianfu$ ali-186590@bef2d:playgrounds dianfu$ docker ps
CONTAINER ID        IMAGE               COMMAND           CREATED          STATUS          PORTS
0880987f0c7        pyflink/playgrounds:1.11.0   "/docker-entrypoint..." 17 seconds ago   Up 16 seconds   6121-6123/tcp, 8081/tcp
6661307b625        playgrounds_taskmanager_1   "playgrounds_taskmanager_1" 17 seconds ago   Up 16 seconds   0.0.0.0:32771->9992/tcp
56fa2deca0212      wurstmeister/kafka_2.11      "start-kafka.sh"    17 seconds ago   Up 16 seconds   0.0.0.0:32771->9992/tcp
0.0.0.0:2181->2181/tcp playgrounds_zookeeper_1   "bin/sh -c 'usr/sbin/wurstmeister/zookeeper/bin/zkServer.sh start'" 19 seconds ago   Up 17 seconds   22/tcp, 2888/tcp, 3888/tcp
719e93da4118        pyflink/playgrounds:1.11.0   "/docker-entrypoint..." 19 seconds ago   Up 17 seconds   6123/tcp, 0.0.0.0:8081->8081
1/tcp               playgrounds_jobmanager_1   "playgrounds_jobmanager_1" 19 seconds ago   Up 17 seconds   0.0.0.0:32771->9992/tcp
ali-186590@bef2d:playgrounds dianfu$ docker-compose exec jobmanager ./bin/flink run -py /opt/examples/1-word_count.py
Job has been submitted with JobID b35c94bd9786680f0c03a1c4380ed
Program execution finished
Job with JobID b35c94bd9786680f0c03a1c4380ed has finished.
Job Runtime: 2826 ms

ali-186590@bef2d:playgrounds dianfu$ cat examples/data/word_count_output/part-e748cc3f-75e2-4a1f-8a82-0b2097536a67-cp-0-task-0-file-0
flink[...]
pyflink[...]
ali-186590@bef2d:playgrounds dianfu$ docker-compose exec jobmanager ./bin/flink run -py /opt/examples/3-udf_add.py
Job has been submitted with JobID 4b7d223be3b5008d0e8bd76d2e1d0a02
Program execution finished
Job with JobID 4b7d223be3b5008d0e8bd76d2e1d0a02 has finished.
Job Runtime: 6842 ms
ali-186590@bef2d:playgrounds dianfu$ cat examples/data/udf_add_output/*
3

```

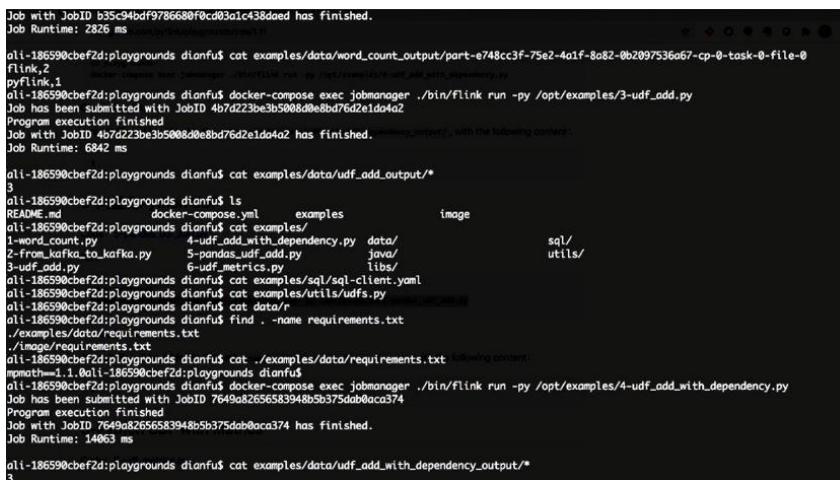
接下来我们再运行一个带有依赖的 Python UDF。前面作业的 UDF 是不包含任何依赖的，直接就把两个输入列相加起来。而在这个例子里，UDF 引用了一个第三方的依赖，我们可以通过 API set python requirement 来执行。



```

  1 from pyflink.table import DataTypes, EnvironmentSettings, BatchTableEnvironment
  2 from pyflink.table.udf import udf
  3
  4 # init environment
  5 environment_settings = EnvironmentSettings.new_instance().use_blink_planner().in_batch_mode().build()
  6 t_env = BatchTableEnvironment.create(environment_settings=environment_settings)
  7 t_env._j_tEnv.getPlanner().getAccrue().setParallelism(1)
  8
  9 t_env.get_config().get_configuration().set_boolean("python.fn-execution.memory.managed", True)
 10
 11 # define and register UDF
 12 @udf(input_types=[DataTypes.BIGINT()], result_type=DataTypes.BIGINT())
 13 def add1(x):
 14     from math import fadd # add third-party dependency
 15     return int(fadd(x, 1))
 16
 17 t_env.register_function("add", add)
 18
 19 # UDF dependency management
 20 t_env.set_python_requirements("/opt/examples/data/requirements.txt")
 21
 22 # Register Source
 23 t_env.execute_sql("""
 24     CREATE TABLE mySource (
 25         a BIGINT,
 26         b BIGINT
 27     ) WITH (
 28         'connector' = 'filesystem',
 29         'format' = 'csv',
 30         'path' = '/opt/examples/data/udf_add_input'
 31     )
 32
 33     
```

接下来我们运行作业，它的执行结果和前面是一样的，因为这两个作业的逻辑是类似的。



```

Job with JobID b35c94bdf9786680f0cd03a1c438doed has finished.
Job Runtime: 2826 ms

ol1-186590cbef2d:playgrounds dianfu$ cat examples/data/word_count_output/part-e748cc3f-75e2-4a1f-8a82-0b2097536a67-cp-0-task-0-file-0
flink_2
ol1-186590cbef2d:playgrounds dianfu$ docker-compose exec jobmanager ./bin/flink run -py /opt/examples/3-udf.add.py
Job has been submitted with JobID 407d223be3b3b5088d0e8bd76d2e1da402
Program execution finished
Job with JobID 407d223be3b3b5088d0e8bd76d2e1da402 has finished.
Job Runtime: 6842 ms

ol1-186590cbef2d:playgrounds dianfu$ cat examples/data/udf_add_output/*
3
ol1-186590cbef2d:playgrounds dianfu$ ls
README.md          docker-compose.yml      examples           image
ol1-186590cbef2d:playgrounds dianfu$ cat examples/
1-word_count.py    4-udf.add_with_dependency.py  data/           sql/
2-from_kafka_to_kafka.py 5-pandas_udf.add.py   java/          utils/
3-udf.add.py       6-udf.metrics.py        libs/          udfs/
ol1-186590cbef2d:playgrounds dianfu$ cat examples/sql/sql-client.yaml
ol1-186590cbef2d:playgrounds dianfu$ cat examples/utils/udfs.py
ol1-186590cbef2d:playgrounds dianfu$ cat data/r
ol1-186590cbef2d:playgrounds dianfu$ find . -name requirements.txt
./examples/data/requirements.txt
./image/requirements.txt
ol1-186590cbef2d:playgrounds dianfu$ cat ./examples/data/requirements.txt
allowing content
maven-w-1.0@ol1-186590cbef2d:playgrounds dianfu$ 
ol1-186590cbef2d:playgrounds dianfu$ docker-compose exec jobmanager ./bin/flink run -py /opt/examples/4-udf.add_with_dependency.py
Job has been submitted with JobID 7649ab2656583948b5b375dbb0aca374
Program execution finished
Job with JobID 7649ab2656583948b5b375dbb0aca374 has finished.
Job Runtime: 14063 ms

ol1-186590cbef2d:playgrounds dianfu$ cat examples/data/udf.add_with_dependency_output/*
3

```

接下来我们再看一个向量化 Python UDF 的例子。在 UDF 定义的时候，我们加了一个 UDF 的 type 字段，说明说我们是一个向量化的 Python UDF，其他的逻

辑和普通 Python UDF 的逻辑类似。最后它的执行结果也是 3，因为它的逻辑和前面是一样的，计算两页的之和。

```

49 lines (36 sloc)  1.2k KB
1 from pyflink.table import DataTypes, EnvironmentSettings, BatchTableEnvironment
2 from pyflink.table.udf import udf
3
4 # Set environment
5 environment_settings = EnvironmentSettings.new_instance().use_blink_planner().in_batch_mode().build()
6 t_env = TableEnvironment.create(environment_settings)
7 t_env.get_config().set_parallelism(2)
8 t_env._j_table.getPlaner().getParallelism();
9
10 t_env.get_config().get_configuration().set_boolean('python.fn-execution.memory.managed', True)
11
12 # Define and register UDF
13 add = udf(lambda l, r: l + r, DataTypes.RECORD(), DataTypes.RECORD(), DataTypes.RECORD(), udf_type='pandas')
14 t_env.register_function("add", add)
15
16 # Register Source
17 t_env.execute_sql("""
18     CREATE TABLE mySource (
19         a BIGINT,
20         b BIGINT
21     ) WITH (
22         'connection' = 'filesystem',
23         'format' = 'csv',
24         'path' = '/opt/examples/data/udf_add_input'
25     );
26 """)
27
28 # Register Sink
29 t_env.execute_sql("""
30     CREATE TABLE mySink (
31         sum BIGINT
32     ) WITH (
33         'connection' = 'filesystem',
34         'format' = 'csv',
35         'path' = '/opt/examples/data/pandas_udf_add_output'
36     );
37 """);

```

我们再来看一个例子，在 Java 的 Table 作业里面使用 Python。在这个作业里面我们又会用到一个 Python UDF，它通过 DDL 语句进行注册，然后在 execute SQL 语句里面进行使用。

```

1 package org.apache.flink.table.api.environment;
2 import org.apache.flink.table.api.TableEnvironment;
3 import org.apache.flink.types.Row;
4
5 import java.util.ArrayList;
6 import java.util.Arrays;
7 import java.util.Iterator;
8 import java.util.List;
9
10 public class BlinkBatchPythonUDFSqlJob {
11     public static void main(String[] args) {
12         TableEnvironment tEnv = TableEnvironment.create(
13             EnvironmentSettings.newInstance().useBlinkPlanner().inBatchMode().build());
14         tEnv.getConfig().getConfiguration().set("parallelOptions", "FAIR,PARALLELISM_1");
15         tEnv.getConfig().getConfiguration().set("python.options.USI.MANAGED_MEMORY", true);
16         tEnv.executeSql("CREATE TEMPORARY SYSTEM FUNCTION add_one AS 'udfs.add_one' LANGUAGE python");
17
18         tEnv.createTemporaryView("source", tEnv.fromValues(1L, 2L, 3L).as("a"));
19
20         DataStream<Row> result = tEnv.executeSql("SELECT add_one(a) AS result FROM source").collect();
21
22         List<Long> actual = new ArrayList();
23         while (result.hasNext()) {
24             Row resultNext = result.next();
25             actual.add(resultNext.getField(0));
26         }
27
28         List<Long> expected = Arrays.asList(2L, 3L, 4L);
29         if (!actual.equals(expected)) {
30             throw new AssertionError(String.format("The output result: %s is not as expected! %s", actual, expected));
31         }
32     }
33 }

```

接下来我们再看在纯 SQL 作业中使用 Python UDF 的例子。在资源文件里面我们声明了一个 UDF，名字叫 add1，它的类型是 Python，同时我们也能看到它的 UDF 位置。

github.com/pylink/playgrounds/blob/1.1/examples/sql/sql-client.yaml

Latest commit [azhais](#) 12 days ago [History](#)

dianfu Add examples 7 and 8.

All 1 contributor

23 lines (21 sloc) 398 Bytes

Raw Blame ⌂ ⌃

```
table:
  - name: sine
    type: sine_table
    update-mode: append
    schema:
      - name: a
        type: BIGINT
    connector:
      type: filesystem
      path: "/opt/examples/data/sql-test-out/result.csv"
  format:
    type: csv
    fields:
      - name: a
        type: BIGINT
  function:
    - name: add_one
      from: python
      fully-qualified-name: udfs.add_one
  configuration:
    python_fn-execution.memory-managed: true
```

接下来我们运行它，执行结果是 234。

```
[INFO] Replacing /usr/local/apache-flink/flink-python_2.11/jar:flink-1.11.0 in the shaded jar.
[INFO] Replacing /Users/dianfu/code/src/github.com/playgrounds/examples/java/target/pyFlink-playgrounds.jar with /Users/dianfu/code/src/github.com/playgrounds/examples/java/target/pyFlink-playgrounds_2.11-0.0.1-shaded.jar
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 7.567 s
[INFO] Finished at: 2020-11-11T16:59:39+08:00
[INFO] Final Memory: 52M/493M
[INFO] -----
[INFO] 
[INFO] -186590cbe2d:java dianfu$ cd -
[INFO] /Users/dianfu/code/src/github.com/playgrounds
[INFO] -186590cbe2d:playgrounds dianfu$ docker-compose exec jobmanager ./bin/flink run -j /opt/examples/java/target/pyFlink-playgrounds.jar -c
[INFO] -186590cbe2d:playgrounds dianfu$ docker-compose exec jobmanager ./bin/flink run -j /opt/examples/java/target/pyFlink-playgrounds.jar -c
[INFO] Job has been submitted with JobID 627e7f9c8db4d078b1b620a4cd6431c5
[INFO] -186590cbe2d:playgrounds dianfu$ docker-compose exec jobmanager ./bin/sql-client.sh embedded --environment /opt/examples/sql/sql-client.yaml --p
[INFO] -186590cbe2d:playgrounds dianfu$ docker-compose exec jobmanager ./bin/sql-client.sh embedded --environment /opt/examples/sql/sql-client.yaml --p
[INFO] No default environment specified.
[INFO] Searching for '/opt/Flink/conf/sql-client-defaults.yaml'...found.
[INFO] Reading default environment from: file:/opt/Flink/conf/sql-client-defaults.yaml
[INFO] Reading session environment from: file:/opt/examples/sql/sql-client.yaml

Command history file path: /root/.flink-sql-history
[INFO] Executing the following statement:
[INFO] insert into sink select add_one(o) from (VALUES (1), (2), (3)) as source (o)
[INFO] Submitting SQL update statement to the cluster.
[INFO] Table update statement has been successfully submitted to the cluster:
Job ID: 4459d20fb9e1a7c7379d4fd0097cb5

[INFO] Shutting down the session...
done.
[INFO] -186590cbe2d:playgrounds dianfu$ cat examples/data/sql-test-out/result.csv
```

四、PyFlink 下一步规划

目前 PyFlink 只支持了 Python Table API，我们计划在下一个版本中支持 DataStream API，同时也会支持 Python UDAF 以及 Pandas UDAF，另外，在执行层也会持续优化 PyFlink 的执行效率。



PyFlink下一步规划

- Python DataStream API
- Python UDAF支持 / Pandas UDAF支持
- Java/Python通信优化

这是一些资源的链接，包括 PyFlink 的文档地址。

- Python Table API 文档：

<https://ci.apache.org/projects/flink/flink-docs-master/api/python/>

- PyFlink 文档：

<https://ci.apache.org/projects/flink/flink-docs-master/dev/table/python/>

- PyFlink playground:

<https://github.com/pyflink/playgrounds/tree/1.11>

Flink Ecosystems

作者：李锐

Apache Flink PMC，阿里巴巴技术专家

本文主要介绍了 Flink SQL 连接外部系统的原因和原理，介绍了常用的 Flink SQL Connector，包括 Kafka Connector、Elasticsearch Connector、FileSystem Connector、Hive Connector 等等。本文主要分为 2 个部分：

- Flink SQL 连接外部系统的实现原理
- Flink SQL 常用的 Connector

一、Flink SQL 连接外部系统的实现原理

在讲原理之前，我们先回答为什么要使用 Flink SQL？SQL 是一个标准化的数据查询语言，而在 Flink SQL 中，我们可以通过 Catalog 与各种系统集成，同时我们也开发了很丰富的内置操作符和函数，而且 Flink SQL 还可以同时处理批数据和流数据，能极大地提高数据分析的工作效率。

那么 Flink SQL 为什么又要对接外部系统呢？Flink SQL 本身是一个流计算的引擎，它本身不维护任何数据，所以对 Flink SQL 而言，所有的数据都存储在外部系统当中，也就是所有的表都是在外部系统中，我们只有对接这些外部系统，才能够对数据进行实际的读写。



Flink SQL对接外部系统

- 何时该使用SQL
 - 通过Catalog方便与各种系统集成
 - 丰富的内置操作符和函数
 - 同一套查询处理批和流数据
- 何时不该使用SQL
 - 需要控制状态和窗口的情况，如自定义timer等
 - 跨版本的savepoint兼容性目前无法保证
- 表总是存储在外部系统中

在讲解 Flink SQL 如何与外部系统对接之前，我们先看一下 Flink 内部 DataStream 和 Table 是如何做转换的？假设已经有一个 DataStream 程序了，那么我们可以把它转换成 Table 的方式来使用，用 Flink SQL 的一些强大功能对它进行查询，可以通过下列例子理解，类似于 Flink SQL 内部的对接。



DataStream与Table的转换

```

StreamTableEnvironment streamTableEnv = StreamTableEnvironment.create(execEnv);
DataStream<Tuple3<String, Long, Double>> sensorData = ...;

// register DataStream
streamTableEnv.createTemporaryView("sensorData", sensorData, $("location"), $("rowtime").rowtime(), $("tempF"));

// query registered Table
Table avgTempCTable = streamTableEnv.sqlQuery(
    "SELECT " +
        "TUMBLE_START(TUMBLE(rowtime, INTERVAL '1' DAY) AS day," +
        "location," +
        "AVG(tempF - 32) * 0.556) AS avgTempC" +
    "FROM sensorData" +
    "WHERE location LIKE 'room%" +
    "GROUP BY location, TUMBLE(rowtime, INTERVAL '1' DAY))");

// go back to DataStream API
DataStream<Row> avgTempC = streamTableEnv.toAppendStream(avgTempCTable, Row.class);

```

(一) Connector

对于 Flink SQL 而言，对接外部系统的组件被称作 Connector。下面这张表里列出了 Flink SQL 所支持的几个比较常用的 Connector，比如 Filesystem 对接的是文件系统，JDBC 对接的是外部的关系型数据库等等。每一个 Connector 主要负责实现一个 source 和一个 sink，source 负责从外部系统中读数据，sink 负责把数据写入到外部系统中。

对接外部系统—Connector



Name	Version	Source	Sink
Filesystem		Bounded and Unbounded Scan, Lookup	Streaming Sink, Batch Sink
Elasticsearch	6.x & 7.x	Not supported	Streaming Sink, Batch Sink
Apache Kafka	0.10+	Unbounded Scan	Streaming Sink, Batch Sink
JDBC		Bounded Scan, Lookup	Streaming Sink, Batch Sink
Apache HBase	1.4.x	Bounded Scan, Lookup	Streaming Sink, Batch Sink

(二) Format

Format 指定了数据在外部系统中的格式，比如一个 Kafka 的表，它里面的数据可能是 CSV 格式存储的，也有可能是 JSON 格式存储的，所以我们在指定一个 Connector 连接外部表的时候，通常也需要指定 Format 是什么，这样 Flink 才能正确地去读写这个数据。

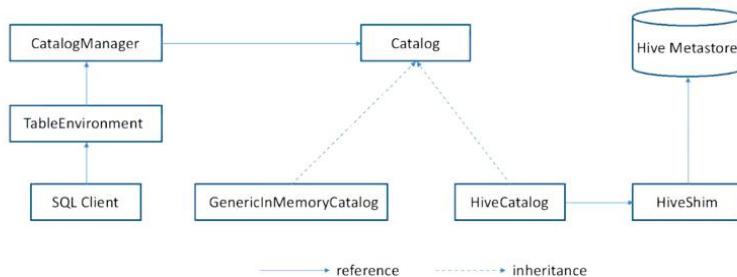
对接外部系统--Format

Formats	Supported Connectors
CSV	Apache Kafka, Filesystem
JSON	Apache Kafka, Filesystem, Elasticsearch
Apache Avro	Apache Kafka, Filesystem
Debezium CDC	Apache Kafka
Canal CDC	Apache Kafka
Apache Parquet	Filesystem
Apache ORC	Filesystem

(三) Catalog

Catalog 可以连接外部系统的元数据，然后把元数据信息提供给 Flink，这样 Flink 可以直接去访问外部系统中已经创建好的表或者 database 等等。比如 Hive 的元数据是存储在 Hive Metastore 中的，那么 Flink 如果想访问 Hive 表的话，就有一个 HiveCatalog 来对接元数据。除此之外，它还可以帮助 Flink 来持久化它自身的元数据。比如说 HiveCatalog 既可以帮 Flink 来访问 Hive，也可以帮 Flink 来存储一些 Flink 所创建的表的信息，这样就不需要每次启动 Session 的时候重新建表了，直接去读取 Hive Metastore 中建好的表就可以了。

对接外部系统--Catalog



如何创建一张表来指定外部的 connector? 下面的例子是通过 DDL 来创建的一张表, 这是一个比较标准的 Create Table 语句, 其中所有跟 Connector 相关的参数都在 with 语句当中指定, 比如这里的 Connector 等于 Kafka 等等。



通过DDL建表

```
CREATE TABLE MyUserTable (
    -- declare the schema of the table
    `user` BIGINT,
    message STRING,
    ts TIMESTAMP,
    proctime AS PROCTIME(), -- use computed column to define proctime attribute
    WATERMARK FOR ts AS ts - INTERVAL '5' SECOND -- use WATERMARK statement to define rowtime attribute
) WITH (
    -- declare the external system to connect to
    'connector' = 'kafka',
    'topic' = 'topic_name',
    'scan.startup.mode' = 'earliest-offset',
    'properties.bootstrap.servers' = 'localhost:9092',
    'format' = 'json' -- declare a format for this system
)
```

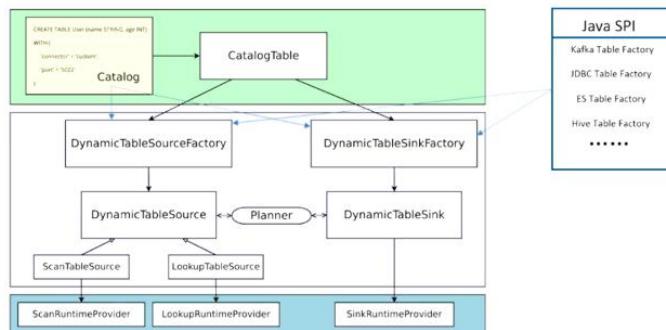
Apache Flink Community China | 极客训练营

当通过 DDL 创建了一张表后, 这个表是如何在 Flink 当中被使用的? 这里有一个很关键的概念就是 Table Factory。在这个黄色的框里面, 我们可以通过 DDL 建表, 或者可以通过 Catalog 从外部系统中拿到, 然后被转化成 Catalog Table 对象。当我们在 SQL 语句中引用 Catalog Table 时, Flink 会为这张表创建对应的 source 或者是 sink, 创建 source 和 sink 的这个模块儿就叫做 Table Factory。

获取 Table Factory 的方式有两种, 一个是 Catalog 本身绑定了一个 Table Factory, 另一种是通过 Java 的 SPI 来确定 Table Factory, 但是它查找的时候要正好有一个配对才不会报错。



Table Factory



二、Flink SQL 常用的 Connector

(一) Kafka Connector

Kafka Connector 是用得最多的，因为 Flink 是一个流计算的引擎，而 Kafka 又是最流行的消息队列，所以用 Flink 的用户大部分也都在用 Kafka。如果我们要创建 Kafka 的表，就需要指定一些特定的参数，比如将 Connector 字段指定成 Kafka，还有 Kafka 对应的 topic 等，我们可以在下图看到这些参数及其所代表的的含义。



Kafka Connector

Option	Required	Default	Type	Description
connector	required	(none)	String	Specify what connector to use, for Kafka the options are: 'kafka', 'kafka-0.11', 'kafka-0.10'.
topic	required	(none)	String	Topic name from which the table is read.
properties.bootstrap.servers	required	(none)	String	Comma separated list of Kafka brokers.
properties.group.id	required by source	(none)	String	The id of the consumer group for Kafka source, optional for Kafka sink.
format	required	(none)	String	The format used to deserialize and serialize Kafka messages. The supported formats are 'csv', 'json', 'avro', 'debezium-json' and 'canal-json'. Please refer to Formats page for more details and more format options.
scan.startup.mode	optional	group-offsets	String	Startup mode for Kafka consumer, valid values are 'earliest-offset', 'latest-offset', 'group-offsets', 'timestamp' and 'specific-offsets'. See the following Start Reading Position for more details.

要使用 Kafka Connector，就需要添加 Kafka 一些依赖的 Jar 包，根据所使用的 Kafka 版本不一样，添加的 Jar 包也不太一样，这些 Jar 包都可以在官网上下载到。



Kafka Dependencies

Kafka Version	Maven dependency	SQL Client JAR
universal	flink-connector-kafka_2.11	Download
0.11.x	flink-connector-kafka-0.11_2.11	Download
0.10.x	flink-connector-kafka-0.10_2.11	Download

(二) Elasticsearch Connector

Elasticsearch Connector 只实现了 Sink，所以只能往 ES 里去写，而不能从里面读。它的 Connector 类型可以指定成 ES6 或者 ES7；Hosts 就是指定的 ES 的各

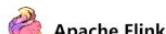
个节点，通过域名加端口号的形式；Index 是指定写 ES 的 index，类似于传统数据库当中的一张表；Document Type 类似于传统数据库的表里面的某一行，不过在 ES7 里不需要指定。



Elasticsearch Connector

Option	Required	Default	Type	Description
connector	required	(none)	String	Specify what connector to use, valid values are: <ul style="list-style-type: none">• elasticsearch-6: connect to Elasticsearch 6.x cluster• elasticsearch-7: connect to Elasticsearch 7.x and later versions cluster
hosts	required	(none)	String	One or more Elasticsearch hosts to connect to, e.g. 'http://host_name:9092;http://host_name:9093'.
index	required	(none)	String	Elasticsearch index for every record. Can be a static index (e.g. 'myIndex') or a dynamic index (e.g. 'index-{log_ts yyyy-MM-dd}'). See the following Dynamic Index section for more details.
document-type	required	(none) in 6.x	String	Elasticsearch document type. Not necessary anymore in elasticsearch-7.

ES 的 Sink 支持 append 和 upsert 两种模式，如果这张 ES 表在定义的时候指定了 PK，那么 Sink 就会以 upsert 模式工作，如果没有指定 PK，就以 append 模式来工作，但是像 ROW 和 MAP 等类型是不能作为 PK 的。



Key Handling

- Elasticsearch的sink支持append模式和upsert模式
- 目标表定义了PK—upsert模式
- 目标表没有PK—append模式
- Upsert模式时，用所有的PK生成document ID
 - 用户可以指定key delimiter
- 某些类型的列不能作为PK
 - ROW、MAP等

同样，使用 ES 也需要指定额外的依赖，针对不同的 ES 版本添加 ES Connector。

Elasticsearch Dependencies

Elasticsearch Version	Maven dependency	SQL Client JAR
6.x	flink-connector-elasticsearch6_2.11	Download
7.x and later versions	flink-connector-elasticsearch7_2.11	Download

(三) FileSystem Connector

这个 Connector 对接的是一个文件系统，它读写的是这个文件系统上的文件。这里所说的 FileSystem 指的是 Flink 的 FileSystem 抽象，它支持很多种不同的实现，比如支持本地文件系统、Hadoop、S3、OSS 等不同的实现。同时它还支持分区，采取与 Hive 相似的分区目录结构，但分区信息不需要注册到 Catalog 中。

FileSystem Connector

- 对应 Flink 的 FileSystem 抽象
 - 支持 Local、Hadoop、S3、OSS 等
- 支持分区
 - 采用与 Hive 相似的分区目录结构
 - 分区信息不需要注册到 Catalog 中
- 支持 Streaming Sink，暂不支持 Streaming Source
- FileSystem Connector 本身不需要添加额外的依赖
 - 可能需要特定 FileSystem 实现的依赖
 - 可能需要特定 Format 的依赖

(四) Hive Connector

Hive 应该是最早的 SQL 引擎，在批处理场景中大部分用户都在使用。Hive Connector 可以分为两个层面，首先在元数据上，我们通过 HiveCatalog 来对接 Hive 元数据，同时我们提供 HiveTableSource、HiveTableSink 来读写 Hive 的表数据。

Hive Connector

- 通过 HiveCatalog 对接 Hive 元数据
- 通过 HiveTableSource、HiveTableSink 读写 Hive 表数据
- 与 Hive 的兼容性
 - 支持多 Hive 版本，1.0.0~3.1.2
 - 支持多种文件格式，text, ORC, Parquet, SequenceFile, RCFile 等
 - 支持多种数据类型
 - Hive Dialect 提供 Hive 风格语法
- Hive 批流一体数仓
 - Streaming Sink, Streaming Source, Lookup Join

使用 Hive Connector 需要指定 Hive Catalog，这里是一个例子，展示如何指定 Hive Catalog。

使用 HiveCatalog

```
execution:  
  planner: blink  
  type: streaming  
...  
current-catalog: myhive # set the HiveCatalog as the current catalog of the session  
current-database: mydatabase  
  
catalogs:  
  - name: myhive  
    type: hive  
    hive-conf-dir: /opt/hive-conf # contains hive-site.xml
```

使用 Hive Connector 也需要添加一些额外的依赖，大家可以根据所使用的 Hive 版本来选择对应的 Jar 包。

Hive Dependencies



Metastore version	Maven dependency	SQL Client JAR
1.0.0 - 1.2.2	flink-sql-connector-hive-1.2.2	Download
2.0.0 - 2.2.0	flink-sql-connector-hive-2.2.0	Download
2.3.0 - 2.3.6	flink-sql-connector-hive-2.3.6	Download
3.0.0 - 3.1.2	flink-sql-connector-hive-3.1.2	Download

除了连接外部系统外，我们也有内置的 Connector，它们一方面是帮助新的用户能够尽快地上手，体验 Flink SQL 强大的功能，另一方面也能帮助 Flink 的开发人员做一些代码的调试。

(五) DataGen Connector

DataGen Connector 是一个数据生成器。比如这里创建了一个 DataGen 的表，指定了几个字段。把 Connector 的类型指定成 DataGen，这个时候去读这张表，Connector 会负责生成数据，也就是说数据是生成出来的，并不是事先要存储在某个地方。然后用户可以对 DataGen Connector 做一些比较细粒度的控制，比如可以指定每秒钟生成多少行数据，然后某个字段可以指定它通过 sequence 也就是从小到大来创建，也可以指定通过 random 的方式来创建等等。



DataGen Connector

```

CREATE TABLE datagen (
    f_sequence INT,
    f_random INT,
    f_random_str STRING,
    ts AS localtimestamp,
    WATERMARK FOR ts AS ts
) WITH (
    'connector' = 'datagen',
    -- optional options --
    'rows-per-second'='5',
    'fields.f_sequence.kind'='sequence',
    'fields.f_sequence.start'='1',
    'fields.f_sequence.end'='1000',
    'fields.f_random.min'='1',
    'fields.f_random.max'='1000',
    'fields.f_random_str.length'='10'
)

```

(六) Print Connector

Print Connector 提供 Sink 功能，负责把所有的数据打印到标准输出或者标准错误输出上，打印的格式是前面会带一个 row kind。创建 print 的表的时候只需要把 Connector 类型指定成 print 就可以了。



Print Connector

- 打印到std out或std error
- 打印格式: \$row_kind(f0,f1,f2...)

```

CREATE TABLE print_table (
    f0 INT,
    f1 INT,
    f2 STRING,
    f3 DOUBLE
) WITH (
    'connector' = 'print'
)

```

(七) BlackHole Connector

BlackHole Connector 也是一个 Sink，它会丢弃掉所有的数据，也就是说数据写过来它什么都不做就丢掉了，主要是可以用来做性能的测试。创建 BlackHole 你只需要把 Connector 类型指定成 BlackHole 就可以了。



BlackHole Connector

- 丢弃所有数据，类似/dev/null

```
CREATE TABLE blackhole_table (
    f0 INT,
    f1 INT,
    f2 STRING,
    f3 DOUBLE
) WITH (
    'connector' = 'blackhole'
)
```

Demo 可以参考：<https://github.com/flink-china/sql-training/wiki/%E7%94%99%E6%80%81%E4%B8%8E%E5%86%99%E5%85%A5%E5%A4%96%E9%83%A8%E8%A1%A8>

Flink Connector 详解

作者：任庆盛

阿里巴巴研发工程师

关于 Flink Connector 的详解，本文将通过四部分展开介绍：

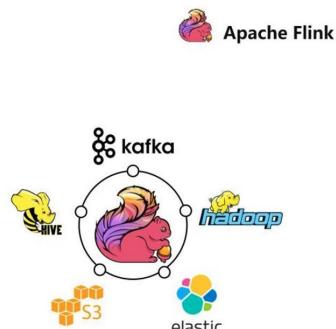
- 连接器
- Source API
- Sink API
- Collector 的未来发展

一、连接器 Connector 的概述-Flink 与外部系统的桥梁

(一) 连接器 Connector

连接器 Connector

- Flink 数据的来源和去向
- 处理流程中的事件控制
 - 事件时间水印 (Watermark)
 - 检查点对齐记录 (Checkpoint Barrier)
- 负载均衡
- 数据解析与序列化
-

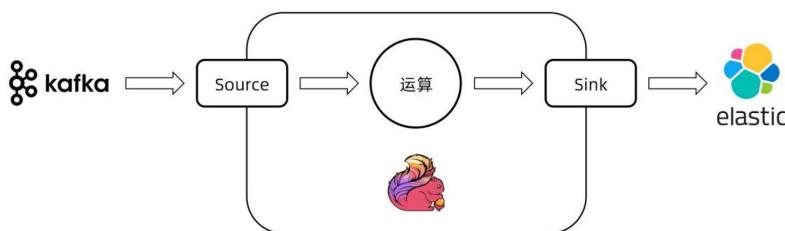


Flink 的数据重要的来源和去向

连接器是 Flink 与外部系统间沟通的桥梁。如：我们需要从 Kafka 里读取数据，在 Flink 里把数据处理之后再重新写回到 HIVE、elastic search 这样的外部系统里去。

- **处理流程中的事件控制：**事件处理水印(watermark)，检查点对齐记录。
- **负载均衡：**根据不同并发的负载对数据分区进行合理的分配。
- **数据解析与序列化：**我们的数据在外部系统里可能是以二进制的形式存储的，在数据库里可能是以各种列的形式来存储的。我们再把它读到 Flink 里后，需要对他进行一个解析，之后才能够进行后面的数据处理。所以我们同样在写回外部系统的时候也需要对数据进行一个序列化的操作-把它转换成外部系统里对应的存储格式来进行存储。

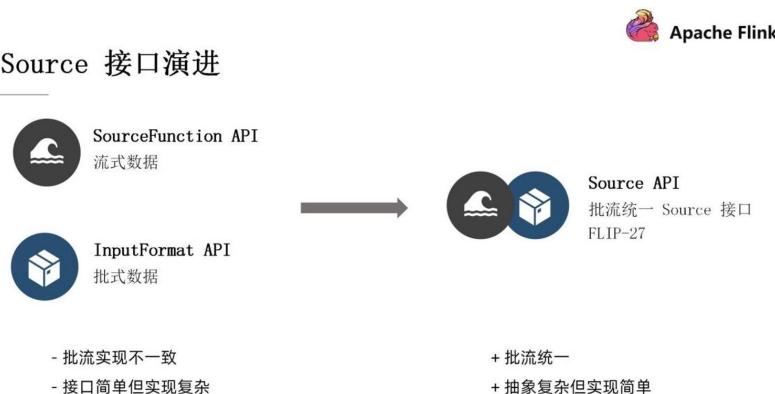
连接器 Connector



我们首先从 kafka 里通过 Source 读取其中的部分记录。然后把这些记录送到 Flink 当中的一些算子进行对应的运算，再通过 Sink 写出到 elastic search 当中去，所以 Source 和 Sink 在这个 Flink 作业的两端起到了一个接口的作用。

二、Source API- Flink 数据的入口

(一) Source 接口演进



Source 在 Flink 1.10 版本之前是左侧的这两个接口：SourceFunction API（用来处理流式数据），InputFormat API（用来处理批式数据）。在 Flink 1.10 之后，社区引入了一个新的 Source API，对整个的 Source 进行了重构。那么为什么我们社区要做这样的一个工作呢？

批流实现不一致：生态不断壮大的过程中，旧的 API 暴露出来一些问题。其中最直观的问题就是批流实现的不一致。

接口简单但实现复杂：之前的 API 可能接口实现比较简单，但实际上对于开发者来讲，在实现这个接口的时候，所有的逻辑、所有的操作实现起来是非常复杂的，对于开发者来讲也不够友好。

因此，基于这些问题，在 FLIP-27 中提出了一个新的 Source API 的设计。其特点有二：

- **批流统一：**流式数据处理和批式数据处理不需要再维护两套代码，用一套代码就够了。
- **实现简单：**Source API 定义了很多概念上的抽象，虽然说这些抽象看起来会比较复杂，但是实际上是简化了开发者操作的开发者开发工作。

(二) 核心抽象



核心抽象



记录分片
Split



记录分片枚举器
SplitEnumerator



Source 读取器
SourceReader

- 有编号的记录集合
- 进度可追踪
- 记录分片的所有信息

- 发现记录分片
- 分配记录分片
- 协调 Source 读取器

- 从记录分片读取数据
- 事件时间水印处理
- 数据解析

1) 记录分片 (Split)

有编号的记录集合

以 Kafka 来举例子。Kafka 的分片既可以定义成一整个分区，也可以定义成一个分区里的某一部分。比如说我从 offset 为 100 的数据开始消费，到 200 号之间我们定义为一个分片；201~300 定义成另外一个分片，这样也是可以的。只要他是一个记录的集合、我们给他一个唯一的编号，我们就可以定义这样的一个记录分片。

进度可追踪

我们需要在这个分片当中记录现在处理到了哪一个位置，我们在记录检查点的时候需要知道当前处理了哪些东西，便于一旦出现了故障，可以直接从故障中恢复起来。

记录分片的所有信息

以 Kafka 举例来讲，一个分区的起始和终止位点等信息是都要包含在整个记录分片里的。因为我们在做 Checkpoint 的时候也是以记录分片为单位的，所以说记录分片里的信息也应该是自洽的。

2) 记录分片枚举器 (Split Enumerator)

发现记录分片：检测外部系统中所存在的分片

分配记录分片：Enumerator 是处于一个协调者的角色存在的。它需要给我们的 Source 读取器分配任务。

协调 Source 读取器：例如某些读取器的进度可能太快了，此时便要告诉他稍微慢一点儿来保证 watermark 大致是一致的。

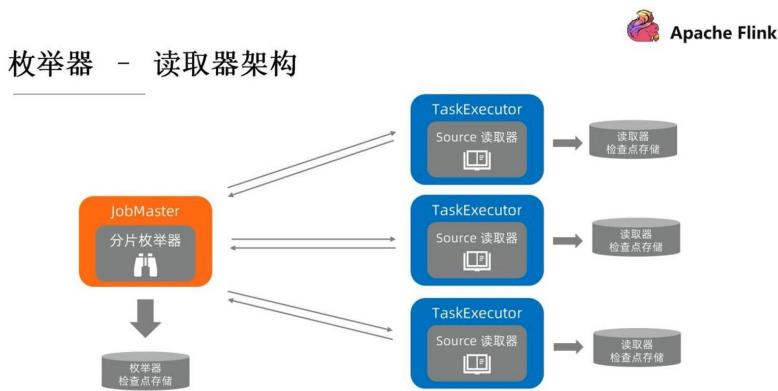
3) Source 读取器 (Source Reader)

从记录分片读取数据：根据枚举器分配的记录分片来读取数据

事件时间水印处理：需要从我们从外部系统中读下来的数据里提取事件时间，然后做出对应的水印发送的操作。

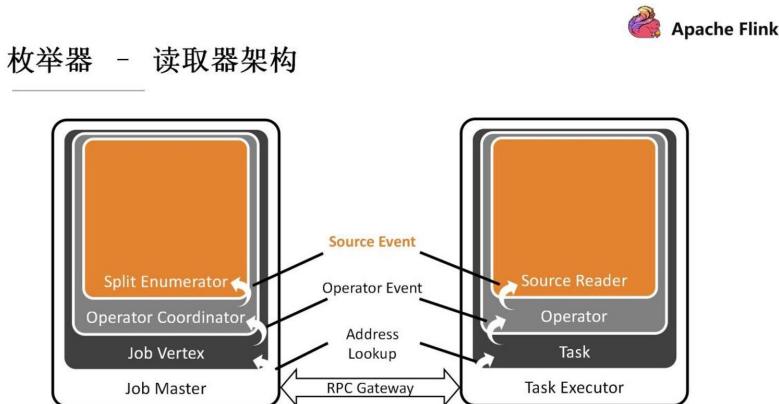
数据解析：对从外部系统中读取到的数据进行反序列化，发送至下游算子

(三) 枚举器-读取器架构



分片枚举器是运行在 Job Master 上面的，Source 读取器是运行在 Task Executor 上面的。因此，枚举器是领导者、协调者的角色，读取器是执行者的角色。

他们的检查点存储也是各自分开的，但之间会存在一些通信。比如说枚举器是需要给读取器来分配任务，也要通知读取器后续没有更多的分片需要处理。由于一个运行环境不一样，他们两个之间也不可避免地会存在一些网络通信。便有了如下通讯栈的定义。



这个通讯栈上面确定了一些 event 来提供给开发者进行自己的实现。

首先，最上面这层是 Source Event，留给开发者自己去定义一些客户化的操作。比如假使现在设计的一个 Source，可能 reader 在某些条件下可能要暂停读取，那么 SplitEnumerator 可以通过这种 Source event 的方式发送给 Source Reader。

其次，再下面一层分别是叫 Operator Coordinator，算子的协调者。它和真正去执行任务的算子通过 Operator Event 算子事件进行沟通的。我们已经事先定义好了

一些算子事件，如添加分片、通知我们的 leader 没有新的分片了等。这些对于所有的 Source 都通用的事件，是在 Operator Event 这一层来进行抽象的。

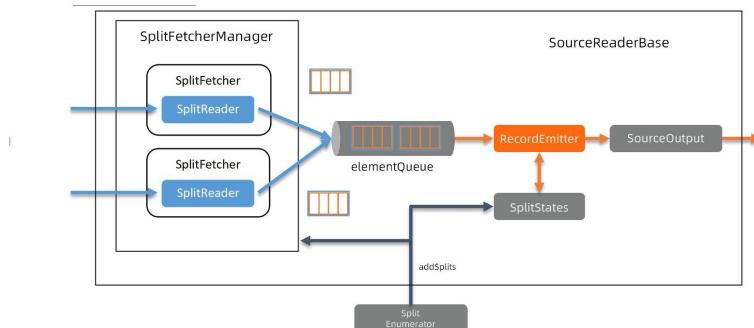
Address Lookup 是用来定位消息应该发送给哪一个 Operator 的。因为 Flink 整个作业执行起来后会有一个加一个有向无环图的。不同的算子可能运行在不同的 Task Manager 上面，那么怎么去找到对应的 task、对应的算子便是这一层的任务。

由于网络通信的存在，Job Master 和 Task Executor 之间有一个 RPC Gateway。所有的 Event 最终都会通过 RPC Gateway、通过 RPC 调用的方式来进行网络传输。

(四) Source 读取器设计

为了简化 Source 读取器实践步骤，减少开发者工作，社区已经为大家提供了 SourceReaderBase。用户在开发的时候可以直接继承 SourceReaderBase 类，从而大大简化开发者的一些开发工作。那么我们接下来对 SourceReaderBase 进行分析。看上去好像这张图里有非常多的组件，但实际上我们可以把它拆成两部分来理解。

Source 读取器设计



以中间 elementQueue 队列作为界限，队列左侧用蓝色标出来的部分是需要和外部系统打交道的组件，在 elementQueue 的右侧用橙色标出来的部分是和 Flink 的引擎侧打交道的部分。

首先，左侧是由一个或者是多个分片的读取器构成的，每一个 reader 通过一个 Fetcher 来驱动，多个 Fetcher 会统一由一个 Fetcher Manager 来管理。这里的实现也有非常多种，比如说可以只开一个线程、只开这一个 SplitReader，通过这一个读取器来消费多个分区。此外，我们也可以根据需求，开多个线程-一个线程运行一个 feature，进行一个 reader，每个 reader 负责一个分区来并行的去消费数据。这些完全取决于用户的实现、选择。

出于性能考虑，每次 SplitReader 会从外部系统中取一批数据，把它们放到 elementQueue 里。如图所示，在这个蓝色框子里的是每次取下来的一批数据，而后橙色框是这一批数据下面的每条数据。

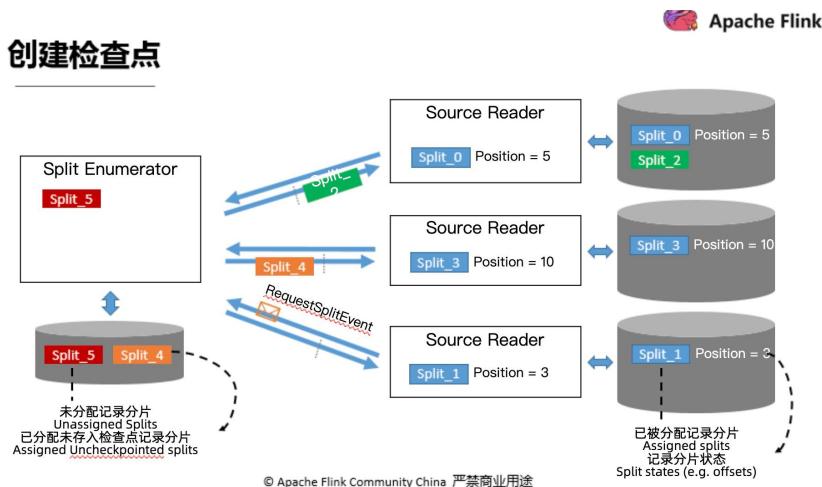
其次，elementQueue 的右侧是由 RecordEmitter 和 SourceOutput 组成的。RecordEmitter 把每条记录发送给下游的另外一个 SourceOutput 会把记录输出出去。每次 RecordEmitter 会从中间 elementQueue 里拿一批数据下来，把它们一条一条发送到下游。由于 RecordEmitter 是由主线程来驱动的，该主线程现在的设计里是用了一个无锁的 mailbox 模型，它会把需要执行工作分成一个一个 mail，每次工作线程从 mailbox 里取出来一个 mail 然后进行工作，所以我们应该注意，这里的实现一定要是无阻塞的。

RecordEmitter 每次往下游发送数据的同时会向下游汇报-后面会不会还有后续的数据需要处理。与此同时呢，我们也会把当前这个分片的处理进度记录在 SplitStates

当中，记录它当前的状态、处理到了什么位置。

当 SplitEnumerator 在外部系统当中发现了新的分片，它需要通过 RPC 调用 addSplits 方法将新的分片添加读取器。在 SplitFetchermanager 这一侧会根据之前用户已经选定的线程模型把新分片分配出去（如只有一个线程，那便会给这个线程分配一个新任务，再让 reader 去读取这个新的分片。如果整体是多线程的实现的，那便新建一个线程，新建一个 reader 来单独去处理分片。同样我们也要在 SplitStates 中记录当前处理的这个进度是怎么样的。

(五) 创建检查点



接下来我们来看一下在新的 Source API 当中是怎么处理检查点的。

首先，左侧我们的协调者，分片枚举器。图中所示，它目前手中还有一个分片 (Split_5) 没有分配出去。中间箭头部分是正在传输路上的一些分片。虚线是这个检查点的边界。我们可以看到二号分片已经在检查点前面了，四号分片在检查点后面，最下方的 reader 正在向 SplitEnumerator 请求一个新的分片。再看 reader，三个 reader 分别已经分配到了某一些 Split、也进行了一些处理，已经有 Position 了。

那我们分别来看一下枚举器和读取器需要在检查点的时候存储哪些东西？

- 枚举器：未分配记录分片 (Split_5)，已分配未存入检查点记录分片 (Split_4)
- 读取器：已被分配记录分片 (Split_0,1,3)，记录分配状态(Split_2)

(六) 三步简单实现 Source

1) Split/SplitState

- Split：外部系统分片
- SplitSerializer：序列化/反序列化 Split 传递给 SourceReader
- SplitState：Split 状态，用于 Checkpoint 与恢复

2) SplitEnumerator

- 发现与订阅 Split
- EnumState：Enumerator 的状态，用于 Checkpoint 与恢复

- EnumStateSerializer: 序列化/反序列化 EnumState

3) SourceReader

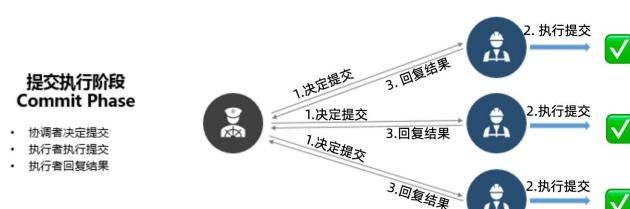
- SplitReader: 与外部系统进行数据交互的接口
- FetcherManager: 选择线程模型 (目前已有)
- RecordEmitter: 转换消息类型与处理事件时间

如果我们仔细去想一下就会发现，其实这些东西绝大多数都是和外部系统打交道的，也就是说和 Flink 引擎本身打交道的部分很少，用户不再需要去担心 checkpoint 锁的问题，多线程的问题等等，能够把更多的开发精力来集中在开发和外部系统交互的部分上。所以说，新的 Source API 是通过这些抽象来大大地简化了开发者的开发。

三、Sink API- Flink 数据的出口

如果对 Flink 有一定的了解的话会发现它可以做到精确一次的语义，数据既不重复也不丢失。那么为了实现这个“精确一次” Flink 也做了很多的工作，其中非常重要的一点就是在 Sink 端实现了二阶段提交。

二阶段提交



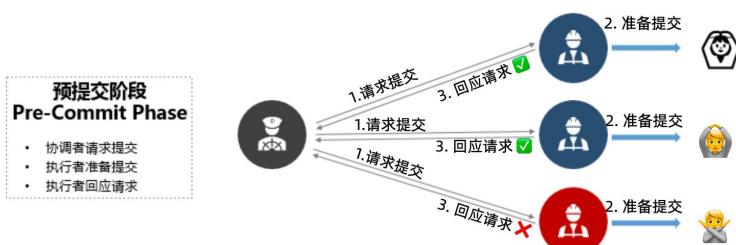
(一) 预提交阶段

在预提交阶段里，由于我们的这个分布式系统一般是存在这种“协调者 1+执行者 n”的模式，那么在预提交的预提交阶段里，首先我们的协调者是需要请求提交的，也就是说他需要给所有的执行者来发送请求提交的消息，从而来开始整个的二阶段提交。

当执行者收到了请求提交的消息，他会做一些提交的准备工作。在所有的准备工作都做完之后，他所有的执行者会向这个协调者回复说明现在已经准备好进行下一步的提交工作了。当协调者收到了所有执行者的“可继续”请求后，预提交阶段结束，进入我们提交第二阶段-提交执行阶段。

(二) 提交执行阶段

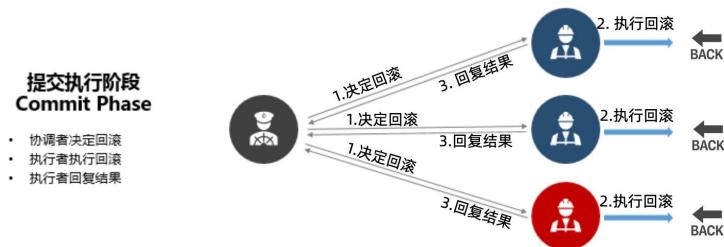
二阶段提交



提交者会向执行者发送决定提交的消息，执行者会把刚刚准备好的提交相关的东西来进行一个处理，来真正的去执行一个提交的动作。在完成之后会向协调者汇报一个回复的结果，反馈提交是否正常执行。

一旦协调者决定进入第二个提交执行阶段，所有的执行者必须要不打折扣地把命令执行下去。也就是说如果某个协调者在这一阶段出了问题的话，他在恢复起来之后还是要把这个决定执行下去的。也就是说一旦决定提交，执行者便必须要把提交这一动作贯彻下去。

二阶段提交

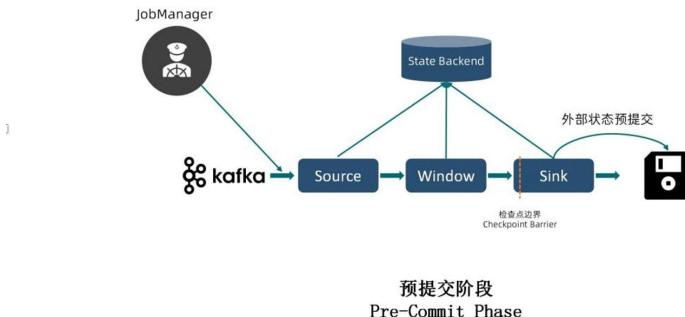


如果在预提交阶段某一个执行者准备提交的时候可能出现了一些故障等、没有做正确的提交动作，那么他可能向协调者会回应了一个错误，比如网络断了，也可能经过一段时间超时之后协调者没有收到这个三号执行者的回应请求，那么协调者就会触发第二阶段的回滚动作。也就是会告诉所有的执行者“这次提交尝试失败了，需要大家回滚到之前的状态”。而后我们的执行者便会出现一个回滚动作，撤销上一步操作。

(三) 二阶段提交在 Flink 中的做法

1) 预提交阶段

二阶段提交

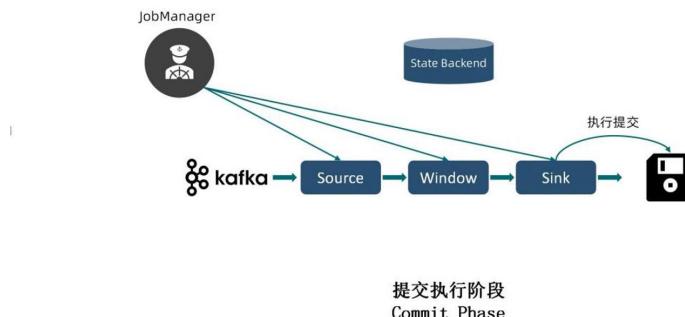


以这个文件系统的 Sink 来举个例子。

文件系统的 Sink 在接收到了检查点边界之后做预提交动作（把当前的数据落盘写到硬盘上的某一个临时文件里），当预提交阶段完成之后，所有的 operator 会向我们的协调者回复“已经准备好进行提交”的信息。

2) 提交执行阶段

二阶段提交



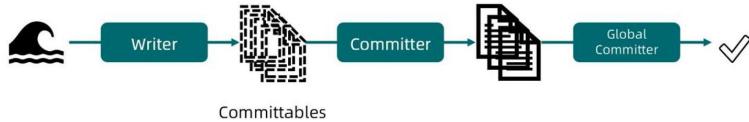
第二个阶段，提交执行阶段开启。JobManager 会向所有的算子发送提交执行的指令，Sink 在接收到这个指令之后，便会真正的去做最后的提交动作。

我们还是以文件系统来举例子，那么刚刚我们已经说过了，在预提交的阶段数据被写到了一个临时文件里，那么在真正的进行提交的时候，临时文件会被按照我们事先定义好的这个名字规范重命名，相当于实现了提交。

这里要注意，临时文件这一设置并非无用，它对后续可能发生的回滚等状况具有铺垫性的作用。我们是巧妙利用了二阶段提交的机制来保障精确一次的语义。

(四) Sink 模型

Sink 模型



Committables
写入 + 预提交阶段 提交阶段 全局提交

- Writer：负责在写入或预提交的阶段，把上游源源不断的的数据写到中间的某一个状态里去。
- Committable：上述所说的“中间的状态”，是可以进行这个提交操作的元件
- Committer：把 Committable 真正的去提交上去
- Global Committer：全局提交器。这个组件是可选的、取决于你的外部系统。
例：Iceberg。

四、未来发展

让连接器开发更简单！



完善新 Source / Sink API



迁移现有连接器至新 API



连接器测试框架

- FLIP-27 Refactor Source Interface:
<https://cwiki.apache.org/confluence/display/FLINK/FLIP-27%3A+Refactor+Source+Interface>
- FLIP-143 Unified Sink API
<https://cwiki.apache.org/confluence/display/FLINK/FLIP-143%3A+Unified+Sink+API>
- An Overview of End-to-End Exactly-Once Processing in Apache Flink
<https://flink.apache.org/features/2018/03/01/end-to-end-exactly-once-apache-flink.html>

完善新 Source

因为 Source 和 Sink 刚刚推出不久，所以说相对来讲还是存在一些问题的。有些开发者可能会有一些新的需求、需要新的更新与提升。目前已经算一个相对稳定的状态，但还是需要去不断地完善。

迁移现有连接器至新 API

随着流批一体连接器的不断推进，所有的连接器会迁移到新的 API 上。

连接器测试框架

连接器测试框架尝试去给所有的 connector 提供一个相对来讲比较一致、统一的测试标准。测试开发者不再需要去自己写一些 case、考虑各种各样的测试环境、测

试场景等等。让我们的开发者能够像搭积木一样快速地用不同的场景，不同的用例来测试自己的代码，从而把更多的开发精力集中在开发这个本身的逻辑上面，大大减少开发者的测试负担。这也是 Source API，Sink API 和后续的 framework 研发的一致目标。是为了让连接器开发更加简单、门槛更低，从而吸引更多的开发者为 Flink 生态做贡献。



Flink 中文社区官方微信



Flink 社区技术交流钉钉群



Flink 中文社区 bilibili 官方账号



阿里云开发者“藏经阁”

海量电子书免费下载