**一文读懂Apache Flink技术**

**一、Flink介绍**

**1.1 Flink基石**

**1.2 Flink API**

**1.3 Flink的用途**

**1.4 Flink Title的变化**

**二、Flink过去与现在**

**2.1 Flink High-Level API的历史变迁**

**2.2 Flink API的历史变迁**

**2.3 Flink Checkpoint & Recovery的历史变迁**

**2.4 Flink Runtime的历史变迁**

**2.5 Flink 网络栈重构**

2.6 Flink 反压

**一、Flink介绍**

Flink是一款分布式的计算引擎，它可以用来做**批处理**，即处理静态的数据集、历史的数据集；也可以用来做**流处理**，即实时地处理一些实时数据流，实时地产生数据的结果；也可以用来做一些**基于事件的应用**，比如说滴滴通过Flink CEP实现实时监测用户及司机的行为流来判断用户或司机的行为是否正当。

总而言之，Flink是一个Stateful Computations Over Streams，即数据流上的有状态的计算。这里面有两个关键字，一个是**Streams**，Flink认为有界数据集是无界数据流的一种特例，所以说有界数据集也是一种数据流，事件流也是一种数据流。Everything is streams，即Flink可以用来处理任何的数据，可以支持批处理、流处理、AI、MachineLearning等等。

另外一个关键词是**Stateful**，即有状态计算。有状态计算是最近几年来越来越被用户需求的一个功能。举例说明状态的含义，比如说一个网站一天内访问UV数，那么这个UV数便为状态。Flink提供了内置的对状态的一致性的处理，即如果任务发生了Failover，其状态不会丢失、不会被多算少算，同时提供了非常高的性能。

那Flink的受欢迎离不开它身上还有很多的标签，其中包括性能优秀（尤其在流计算领域）、高可扩展性、支持容错，是一种纯内存式的一个计算引擎，做了内存管理方面的大量优化，另外也支持eventime的处理、支持超大状态的Job（在阿里巴巴中作业的state大小超过TB的是非常常见的）、支持exactly-once的处理。

**1.1 Flink基石**

Flink之所以能这么流行，离不开它最重要的四个基石：**Checkpoint、State、Time、Window。**

首先是**Checkpoint机制**，这是Flink最重要的一个特性。Flink基于Chandy-Lamport算法实现了一个分布式的一致性的快照，从而提供了一致性的语义。Chandy-Lamport算法实际上在1985年的时候已经被提出来，但并没有被很广泛的应用，而Flink则把这个算法发扬光大了。Spark最近在实现Continue streaming，Continue streaming的目的是为了降低它处理的延时，其也需要提供这种一致性的语义，最终采用Chandy-Lamport这个算法，说明Chandy-Lamport算法在业界得到了一定的肯定。

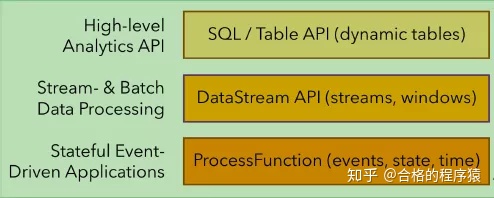
提供了一致性的语义之后，Flink为了让用户在编程时能够更轻松、更容易地去管理状态，还提供了一套非常简单明了的**State API**，包括里面的有ValueState、ListState、MapState，近期添加了BroadcastState，使用State API能够自动享受到这种一致性的语义。

除此之外，Flink还实现了Watermark的机制，能够支持**基于事件的时间**的处理，或者说基于系统时间的处理，能够容忍数据的延时、容忍数据的迟到、容忍乱序的数据。

另外流计算中一般在对流数据进行操作之前都会先进行**开窗**，即基于一个什么样的窗口上做这个计算。Flink提供了开箱即用的各种窗口，比如滑动窗口、滚动窗口、会话窗口以及非常灵活的自定义的窗口。

**1.2 Flink API**

Flink分层API主要有三层，如下图：



最底层是ProcessFunction，它能够提供非常灵活的功能，它能够访问各种各样的State，用来注册一些timer，利用timer回调的机制能够实现一些基于事件驱动的一些应用。

之上是DataStream API，最上层是SQL/Table API的一种High-level API。

**1.3 Flink的用途**

Flink能用来做什么？回顾一下Flink up前几站的分享，有非常多的嘉宾分享了他们在自己公司里面基于Flink做的一些实践，包括携程、唯品会、饿了么、滴滴、头条等等。他们的应用场景包括实时的机器学习，实时的统计分析，实时的异常监测等等。这些实践案例的共同点就是都用来做**实时性**的任务。

**1.4 Flink Title的变化**

早期Flink是这样介绍自己的:“我是一个开源的流批统一的计算引擎”，当时跟Spark有点类似。后来Spark改成了一长串的文字，里面有各种各样的形容词：“我是一个分布式的、高性能的、高可用的、高精确的流计算系统”。最近Spark又进行了修改：“我是一个数据流上的有状态的计算”。

通过观察这个变化，可以发现Flink社区重心的变迁，**即社区现在主要精力是放在打造它的流计算引擎上**。先在流计算领域扎根，领先其他对手几年，然后借助社区的力量壮大社区，再借助社区的力量扩展它的生态。

阿里巴巴Flink是这样介绍自己的：“Flink是一个大数据量处理的统一的引擎”。这个“统一的引擎”包括流处理、批处理、AI、MachineLearning、图计算等等。

**二、Flink过去与现在**

**2.1 Flink High-Level API的历史变迁**

**在Flink 1.0.0时期，**Table API和CEP这两个框架被首次加入到仓库里面，同时社区对于SQL的需求很大。SQL和Table API非常相近，都是一种处理结构化数据的一种High-Level语言，实现上可以共用很多内容。所以在1.1.0里面，社区基于Apache Calcite对整个非Table的Module做了重大的重构，使得Table API和SQL共用了大部分的代码，同时进行了支持。

**在Flink 1.2.0时期，**在Table API和SQL上支持Tumbling Window、Sliding Window、Session Window这些窗口。

**在Flink 1.3.0时期，**首次引用了Dynamic Table这个概念，借助Dynamic Table，流和批之间是可以相互进行转换的。流可以是一张表，表也可以是一张流，这是流批统一的基础之一。Retraction机制是Dynamic Table最重要的一个功能，基于Retraction才能够正确地实现多级Application、多级Join，才能够保证语意与结果的一个正确性。同时该版本支持了CEP算子的可控性。

**在Flink 1.5.0时期，**支持了Join操作，包括window Join以及非window Join，还添加了SQL CLI支持。SQL CLI提供了一个类似shell命令的对话框，可以交互式执行查询。

**2.2 Flink API的历史变迁**

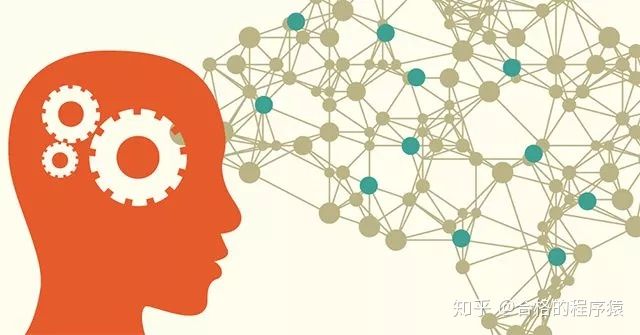
**在Flink 1.0.0时期，**加入了State API，即ValueState、ReducingState、ListState等等。State API主要方便了DataStream用户，使其能够更加容易地管理状态。

**在Flink 1.1.0时期，**提供了对SessionWindow以及迟到数据处理的支持。

**在Flink 1.2.0时期，**提供了ProcessFunction，一个Low-level的API。基于ProcessFunction用户可以比较灵活地实现基于事件的一些应用。

**在Flink 1.3.0时期，**提供了Side outputs功能。一般算子的输出只有一种输出的类型，但是有些时候可能需要输出另外的类型，比如把一些异常数据、迟到数据以侧边流的形式进行输出，并交给异常节点进行下一步处理，这就是Side outputs。

**在Flink 1.5.0时期，**加入了BroadcastState。BroadcastState用来存储上游被广播过来的数据，这个节点上的很多N个并发上存在的BroadcastState里面的数据都是一模一样的，因为它是从上游广播来的。基于这种State可以比较好地去解决不等值Join这种场景。比如一个Query里面写的“SLECECT \* FROM L JOIN R WHERE L.a > R.b”，也就是说我们需要把左表和右表里面所有A大于B的数据都关联输出出来。



在以前的实现中，由于没有Join等值条件，就无法按照等值条件来做KeyBy的Shuffle，只能够将所有的数据全部汇集到一个节点上，一个单并发的节点上进行处理，而这个单并发的节点就会成为整个Job的瓶颈。

而有了BroadcastState以后就可以做一些优化：因为左表数据量比较大，右表数据量比较小，所以选择把右表进行广播，把左表按照它某一个进行均匀分布的key，做keyby shuffle，shuffle到下游的N个Join的节点，Join的节点里面会存两份State，左边state和右边state，左边state用来存左边数据流的state，是一个keyedState，因为它是按它某一个key做keyby分发下来的。右边State是一个BroadcastState，所有的Join节点里面的BroadcastState里面存的数据都是一模一样的，因为均为从上游广播而来。

所有keyedState进行并发处理，之后将keyedState集合进行合并便等于左边数据流的全集处理结果。于是便实现了这个Join节点的可扩充，通过增加join节点的并发，可以比较好地提升Job处理能力。除了不等值Join场景，BroadcastState还可以比较有效地解决像CAP上的动态规则。

**在Flink 1.6.0时期，**提供了State TTL参数、DataStream Interval Join功能。State TTL实现了在申请某个State时候可以在指定一个TTL参数，指定该state过了多久之后需要被系统自动清除。在这个版本之前，如果用户想要实现这种状态清理操作需要使用ProcessFunction注册一个Timer，然后利用Timer的回调手动把这个State清除。从该版本开始，Flink框架可以基于TTL原生地解决这件事情。DataStream Interval Join功能即含有区间间隔的Join，比如说左流Join右流前后几分钟之内的数据，这种叫做Interval Join。

**2.3 Flink Checkpoint & Recovery的历史变迁**

Checkpoint机制在Flink很早期的时候就已经支持，是Flink一个很核心的功能，Flink社区也一直致力于努力把Checkpoint效率提升，以及换成FailOver之后它的Recallable效率的提升。

**在Flink 1.0.0时期，**提供了RocksDB的支持，这个版本之前所有的状态都只能存在进程的内存里面，这个内存总有存不下的一天，如果存不下则会发生OOM。如果想要存更多数据、更大量State就要用到RocksDB。RocksDB是一款基于文件的嵌入式数据库，它会把数据存到磁盘，但是同时它又提供高效读写能力。所以使用RocksDB不会发生OOM这种事情。在Flink1.1.0里面，提供了纯异步化的RocksDB的snapshot。以前版本在做RocksDB的snapshot时它会同步阻塞主数据流的处理，很影响吞吐量，即每当checkpoint时主数据流就会卡住。纯异步化处理之后不会卡住数据流，于是吞吐量也得到了提升。

**在Flink 1.2.0时期，**引入了Rescalable keys和operate state的概念，它支持了一个Key State的可扩充以及operator state的可扩充。

**在Flink 1.3.0时期，**引入了增量的checkpoint这个比较重要的功能。只有基于增量的checkpoint才能更好地支持含有超大State的Job。在阿里内部，这种上TB的State是非常常见。如果每一次都把全量上TB的State都刷到远程的HDFS上那么这个效率是很低下的。而增量checkpoint只是把checkpoint间隔新增的那些状态发到远程做存储，每一次checkpoint发的数据就少了很多，效率得到提高。在这个版本里面还引入了一个细粒度的recovery，细粒度的recovery在做恢复的时候，有时不需要对整个Job做恢复，可能只需要恢复这个Job中的某一个子图，这样便能够提高恢复效率。

**在Flink 1.5.0时期，**引入了Task local 的State的recovery。因为基于checkpoint机制，会把State持久化地存储到某一个远程存储，比如HDFS，当发生Failover的时候需要重新把这个数据从远程HDFS再download下来，如果这个状态特别大那么该download操作的过程就会很漫长，导致Failover恢复所花的时间会很长。Task local state recovery提供的机制是当Job发生Failover之后，能够保证该Job状态在本地不会丢失，进行恢复时只需在本地直接恢复，不需从远程HDFS重新把状态download下来，于是就提升了Failover recovery的效率。



**2.4 Flink Runtime的历史变迁**

Runtime的变迁历史是非常重要的。

**在Flink 1.2.0时期，**提供了Async I/O功能。如果任务内部需要频繁地跟外部存储做查询访问，比如说查询一个HBase表，在该版本之前每次查询的操作都是阻塞的，会频繁地被I/O的请求卡住。当加入异步I/O之后就可以同时地发起N个异步查询的请求，这样便提升了整个job的吞吐量，同时Async I/O又能够保证该job的Async语义。

**在Flink 1.3.0时期，**引入了HistoryServer的模块。HistoryServer主要功能是当job结束以后，它会把job的状态以及信息都进行归档，方便后续开发人员做一些深入排查。

**在Flink 1.4.0时期，**提供了端到端的exactly once的语义保证，Flink中所谓exactly once一般是指Flink引擎本身的exactly once。如果要做到从输入到处理再到输出，整个端到端整体的exactly once的话，它需要输出组件具备commit功能。在kafka老版本中不存在commit功能，从最近的1.1开始有了这个功能，于是Flink很快便实现了端到端exactly once。

**在Flink 1.5.0时期，**Flink首次对外正式地提到新的部署模型和处理模型。新的模型开发工作已经持续了很久，在阿里巴巴内部这个新的处理模型也已经运行了有两年以上，该模型的实现对Flink内部代码改动量特别大，可以说是自Flink项目建立以来，Runtime改动最大的一个改进。简而言之，它的一个特性就是它可以使得在使用YARN、Mesos这种调度系统时，可以更加更好地动态分配资源、动态释放资源、提高资源利用性，还有提供更好的jobs之间的隔离。最后是在这个版本中，Flink对其网络站进行了一个基本重构。

**2.5 Flink 网络栈重构**

在流计算中有两个用来衡量性能的指标：**延迟**和**吞吐**。

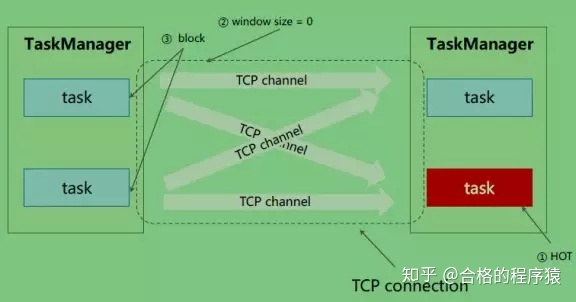
一般来讲如果想要更高吞吐就要牺牲一些延迟，如果想要更低的延迟就要牺牲一定的吞吐。**但是网络栈的重构却实现了延迟和吞吐的同时提升**，这主要得益于它两方面的工作：第一个是基于信用的流控，另一个是基于事件的I/O。一个用来提高它的吞吐，另一个用来降低它的延迟。

在介绍流控之前需要先介绍一下现有的网络栈。Flink中TaskManager就是用来管理各个task的角色，它是以进程为单位；task用来执行用户代码，以线程为单位。当tasks之间有数据传输的交互的时候就要建立网络的连接，如果2秒之间都建立一个TCP连接的话，那么这个TCP连接会被严重浪费，所以Flink在两个TaskManager之间建立一个TCP连接，即两个进程之间只存在一个连接。各个task之间以TCP channel的方式来共享TCP的连接，这样整个job中就不会有太多的TCP连接。

**2.6 Flink 反压**

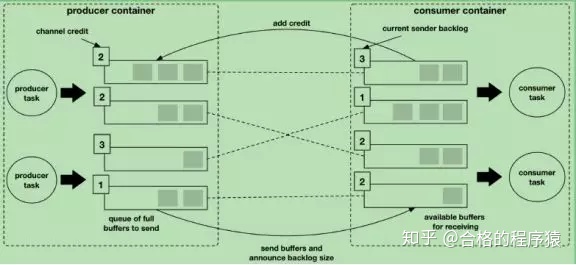
反压的意思是当某一个task的处理性能跟不上输入速率的时候，其输入端的Buffer就会被填满，当输入端Buffer被填满的时候就会导致TCP的读取被暂停。TCP的读取被暂停之后，就会导致上游输出端的Buffer池越积越多，因为下游此时已经不再进行消费。

当上游输出端的Buffer池也堆满的时候， TCP通道就会被关闭，其内部所有的TCP channel也会被关闭。从而上游task就会逐级的向上游进行反压，这是整体的反压流程，所以说Flink以前的反压机制是比较原生态、比较粗暴的，因为其控制力度很大，整个TCP中一旦某一个Task性能跟不上，就会把整个TCP连接关掉。如下图所示：



右下角的task虽然处理跟不上了，但上面的task仍然可以继续进行处理。左边这些上游数据可以继续发给右上角的task进行处理。但是由于现在整个的TCP连接都被关闭，导致右上角task同样收不到数据，整体吞吐量实际上是下降的趋势。**为了优化这个功能就需要做到更加细密度的流控**，目前是关闭整个TCP连接，优化措施就是需要对TCP channel进行控制，当某个task处理不过来时只需要该Task对应的TCP channel，其它TCP channel不受影响。优化实现方式就是基于信用的流控。

**基于信用的流控的核心思想就是基于信用额度的消费。**比如银行做贷款，为了防止坏账太多，它会对每一个人评估其信用额度，当发放贷款时贷款不会超过这个人能承受的额度。基于这种方式，它能够一方面不会产生太多坏账，另一方面可以充分地把银行的资金利用起来。基于信用的流控就是基于这种思想，Flink中所谓的信用额度，就是指这个下游消费端的可用的Buffer数。如下图：



该图左边是指发送端，有四个输出的队列，每个队列里面的方块代表输出Buffer，即准备丢给下游处理的Buffer。右边是消费端，消费端也有四个队列，这四个队列里面也有一些Buffer块，这些Buffer块是空闲的Buffer，准备用来接收上游发给自己的数据。

上面提到基于数据的流控中所谓的信用就是指这个消费端它可用的Buffer数，代表当前还能够消费多少数据，消费端首先会向上游反馈当前的信用是多少， producer端只会向信用额度大于0的下游进行发送，对于信用额度如果为0的就不再发送数据。这样整个网络的利用率便得到了很大的提升，不会发生某些Buffer被长时间的停留在网络的链路上的情况。

**基于信用的流控主要有以下两方面的优化提升：**

**一个是当某一个task发生反压处理跟不上的时候，不会发生所有的task都卡住**，这种做法使吞吐量得到了很大的提升，在阿里内部用双11大屏作业进行测试，这种新的流控算法会得到20%的提升；

**另一个是基于事件的I/O，Flink在网络端写数据时会先往一个Buffer块里面写数据**，这个Buffer块是一个32K的长度的单位，即32K的大小，当这个Buffer块被填满的时候就会输出到网络里面，或者如果数据流比较慢，没办法很快填满的话，那么会等待一个超时，默认一个100毫秒，即如果100毫秒内还没被填满那么这个Buffer也会被输出到网络里面。此时若是在以前版本中Flink延迟可能是在100毫秒以内，最差的情况下是到100毫秒，因为需要到100毫秒等这个Buffer发出去。

如果要得到更低的延时，现在的做法就会将这个Buffer直接加入到输出的队列，但是还是保持继续往这个Buffer块里面写数据，当网络里面有容量时这个Buffer块便会立刻被发出去，如果网络现在也比较繁忙，那就继续填充这个Buffer，这样吞吐也会比较好一点。基于这种算法，Flink的延时几乎是完美的，可以看到它的曲线基本上是低于10毫秒的，这也充分利用了网络的容量，几乎对吞吐没有影响。