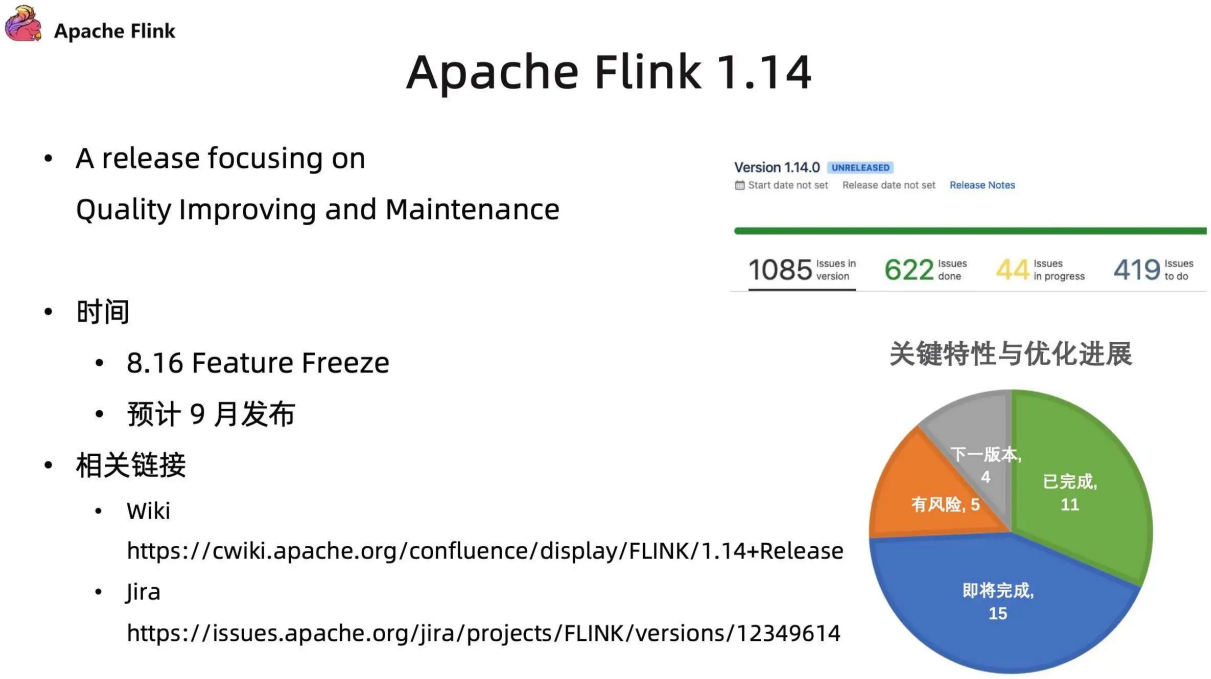
Apache Flink 1.14 新特性介绍

一、简介

1.14 新版本原本规划有 35 个比较重要的新特性以及优化工作，目前已经有 26 个工作完成；5 个任务不确定是否能准时完成；另外 4
个特性由于时间或者本身设计上的原因，会放到后续版本完成。[1]



如果想及时了
解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公共帐号：过往记忆大数据

1.14 相对于历届版本来说，囊括的优化和新增功能点其实并不算多。通过观察发版的节奏可以发现，通常在 1-2
个大版本后都会发布一个变化稍微少一点的版本，主要目的是把一些特性稳定下来。

1.14 版本就是这样一个定位，我们称之为质量改进和维护的版本。这个版本预计 8 月 16 日停止新特性开发，可能在 9
月份能够和大家正式见面，有兴趣可以关注以下链接去跟踪功能发布进度。

- Wiki：https://cwiki.apache.org/confluence/display/FLINK/1.14+Release
- Jira：https://issues.apache.org/jira/projects/FLINK/versions/12349614

[1] 截至到 8 月 31 日，确定进入新版本的是 33 个，已全部完成。

二、流批一体

流批一体其实从 Flink 1.9 版本开始就受到持续的关注，它作为社区 RoadMap 的重要组成部分，是大数据实时化必然的趋势。但是另一方面，传统离线的计算需求其实并不会被实时任务完全取代，而是会长期存在。

在实时和离线的需求同时存在的状态下，以往的流批独立技术方案存在着一些痛点，比如：

- 需要维护两套系统，相应的就需要两组开发人员，人力的投入成本很高；
- 另外，两套数据链路处理相似内容带来维护的风险性和冗余；
- 重要的一点是，如果流批使用的不是同一套数据处理系统，引擎本身差异可能会存在数据口径不一致的问题，从而导致业务数据存在一定的误差。这种误差对于大数据分析会有比较大的影响。

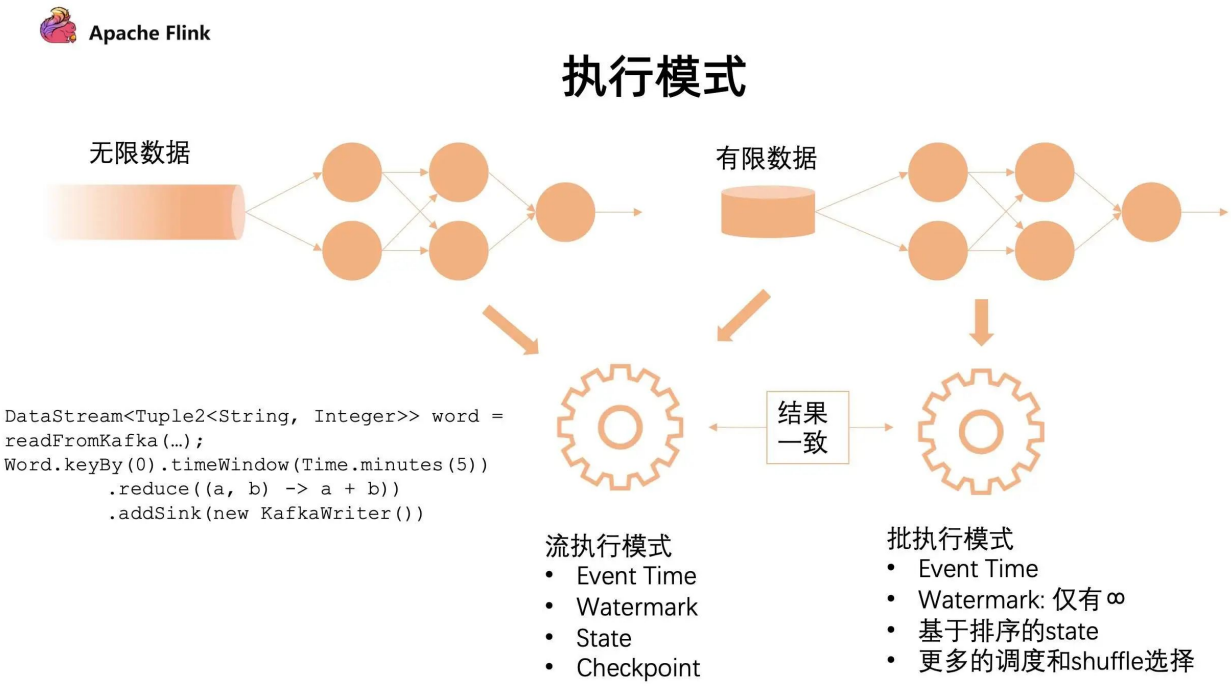
在这样的背景下，Flink 社区认定了实时离线一体化的技术路线是比较重要的技术趋势和方向。

Flink 在过去的几个版本中，在流批一体方面做了很多的工作。可以认为 Flink 在引擎层面，API 层面和算子的执行层面上做到了真正的流与批用同一套机制运行。但是在任务具体的执行模式上会有 2 种不同的模式：

对于无限的数据流，统一采用了流的执行模式。流的执行模式指的是所有计算节点是通过 Pipeline 模式去连接的，Pipeline 是指上游和下游计算任务是同时运行的，随着上游不断产出数据，下游同时在不断消费数据。这种全 Pipeline 的执行方式可以：

- 通过 eventTime 表示数据是什么时候产生的；
- 通过 watermark 得知在哪个时间点，数据已经到达了；
- 通过 state 来维护计算中间状态；
- 通过 Checkpoint 做容错的处理。

下图是不同的执行模式：



如果想及时了
解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

对于有限的数据集有 2
种执行模式
，我们可以把它看成一个有限的数据流去做处理，也可以把它看成批的执行模式。批的执行模式虽然也有 eventTime，但是对于 watermark 来说只支持正无穷。对数据和 state
排序后，它在任务的调度和 shuffle 上会有更多的选择。

流批的执行模式是有区别的，最主要的就是批的执行模式会有落盘的中间过程，只有当前面任务执行完成，下游的任务才会触发，这个容错机制是通过 shuffle 进行容错的。

这 2 者也各有各的执行优势：

- 对于流的执行模式来说，它没有落盘的压力，同时容错是基于数据的分段，通过不断对数据进行打点 Checkpoint 去保证断点恢复；
- 然而在批处理上，因为要经过 shuffle 落盘，所以对磁盘会有压力。但是因为数据是经过排序的，所以对于批来说，后续的计算效率可能会有一定的提升。同时，在执行时候是经过分段去执行任务的，无需同时执行；在容错计算方面是根据 stage 进行容错。

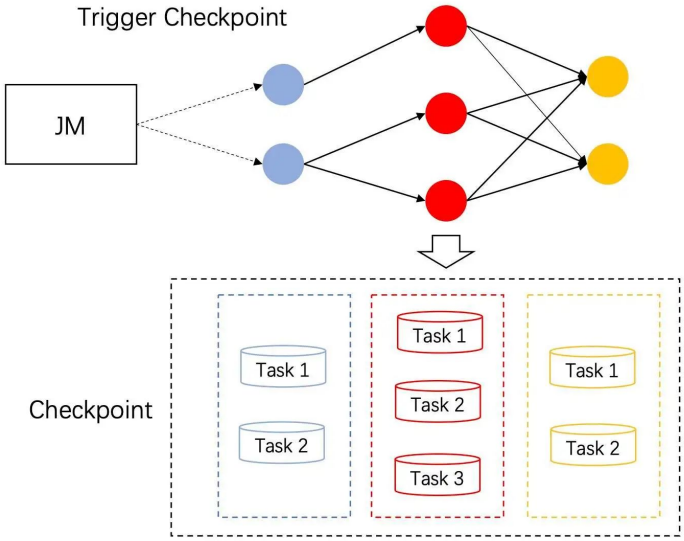
这两种各有优劣，可以根据作业的具体场景来进行选择。

Flink 1.14 的优化点主要是针对在流的执行模式下，如何去处理有限数据集。之前处理无限数据集，和现在处理有限数据集最大的区别在于引入了 "任务可能会结束"
的概念。这种情况下带来了一些新的问题，如下图:

Apache Flink

流执行模式的 Checkpoint

- 无限流
  - 从 Source 触发
  - 包含所有 Task 的 State



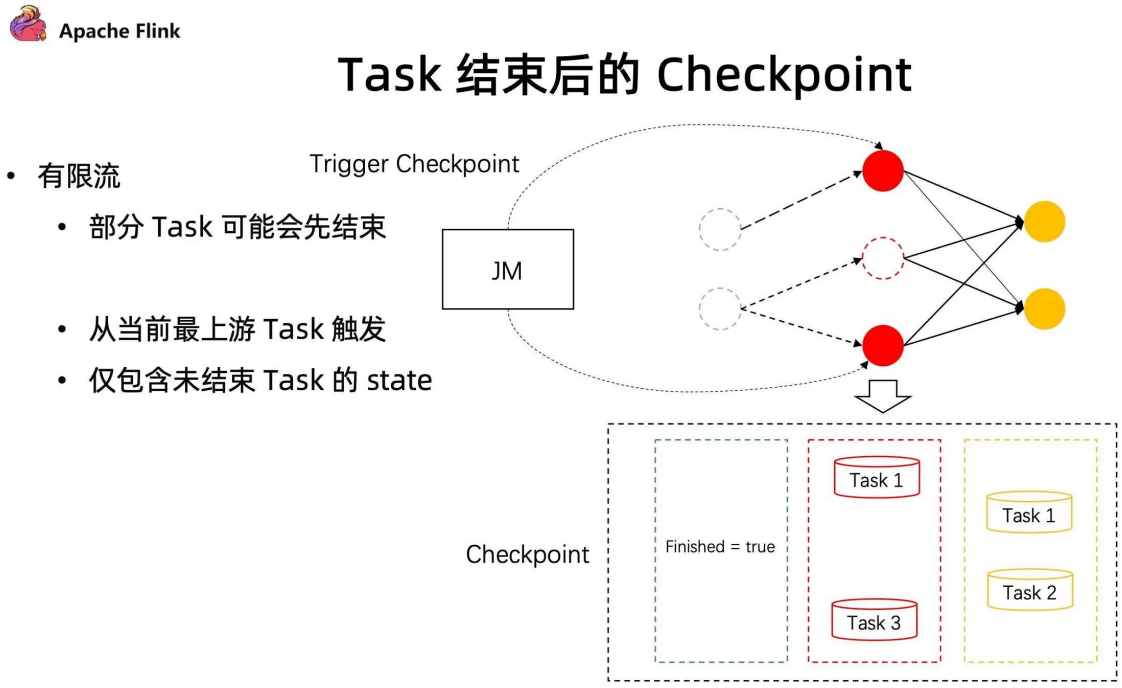
如果想及时了
解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

在流的执行模式下的 Checkpoint 机制

对于无限流，它的 Checkpoint 是由所有的 source 节点进行触发的，由 source 节点发送 Checkpoint Barrier ，当 Checkpoint Barrier 流过整个作业时候，同时会存储当前作业所有的 state 状态。

而在有限流的 Checkpoint 机制中，Task 是有可能提早结束的。上游的 Task 有可能先处理完任务提早退出了，但下游的 Task 却还在执行中。在同一个 stage 不同并发下，有可能因为数据量不一致导致部分任务提早完成了。这种情况下，在后续的执行作业中，如何进行 Checkpoint ？

在 1.14 中，JobManager 动态根据当前任务的执行情况，去明确 Checkpoint Barrier 是从哪里开始触发。同时在部分任务结束后，后续的 Checkpoint 只会保存仍在运行 Task 所对应的 stage，通过这种方式能够让任务执行完成后，还可以继续做 Checkpoint ，在有限流执行中提供更好的容错保障。



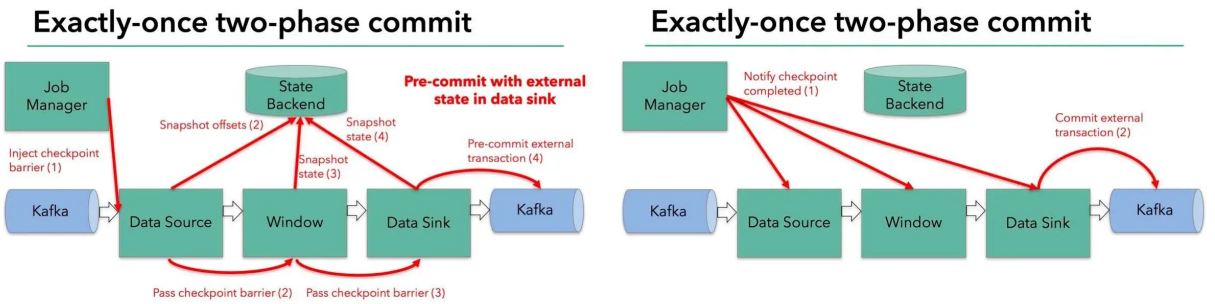
如果想及时了
解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

Task 结束后的两阶段提交

我们在部分 Sink 使用上，例如下图的 Kafka Sink 上，涉及到 Task 需要依靠 Checkpoint 机制，进行二阶段提交，从而保证数据的 Exactly-once 一致性。

Apache Flink

Task 结束后的两阶段提交



https://flink.apache.org/features/2018/03/01/end-to-end-exactly-once-apache-flink.html

- 有限流
  - Task / 作业结束后不能保证有 Checkpoint，最后一部分数据如何提交?
  - 数据处理完成后，Task 等待 Checkpoint 完成后再退出

具体可以这样说：在 Checkpoint 过程中，每个算子只会进行准备提交的操作。比如数据会提交到外部的临时存储目录下，所有任务都完成这次 Checkpoint
后会收到一个信号，之后才会执行正式的
commit，把所有分布式的临时文件一次性以事务的方式提交到外部系统。

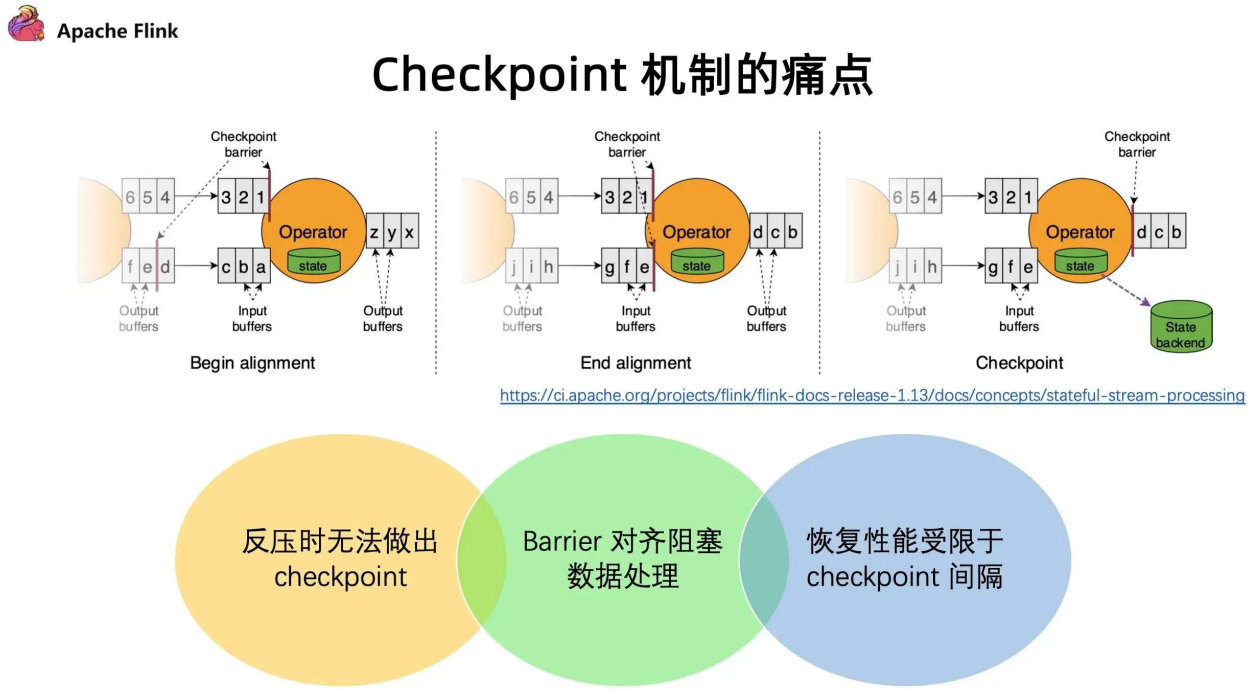
这种算法在当前有限流的情况下，作业结束后并不能保证有
Checkpoint，那么最后一部分数据如何提交？

在 1.14 中，这个问题得到了解决。Task 处理完所有数据之后，必须等待 Checkpoint 完成后才可以正式的退出，这是流批一体方面针对有限流任务结束的一些改进。

三、checkpoint 机制

现有 Checkpoint 机制痛点

目前 Flink 触发 Checkpoint 是依靠 barrier 在算子间进行流通，barrier
随着算子一直往下游进行发送，当算子下游遇到 barrier 的时候就会进行快照操作，然后再把 barrier 往下游继续发送。对于多路的情况我们会把 barrier 进行对齐，把先到 barrier
的这一路数据暂时性的 block，等到两路 barrier 都到了之后再做快照，最后才会去继续往下发送 barrier。



如果想及时了
解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

现有的 Checkpoint 机制存在以下问题：

- 反压时无法做出 Checkpoint ：在反压时候 barrier
  无法随着数据往下游流动，造成反压的时候无法做出
  Checkpoint。但是其实在发生反压情况的时候，我们更加需要去做出对数据的
  Checkpoint，因为这个时候性能遇到了瓶颈，是更加容易出问题的阶段；
- Barrier 对齐阻塞数据处理 ：阻塞对齐对于性能上存在一定的影响；
- 恢复性能受限于 Checkpoint 间隔
  ：在做恢复的时候，延迟受到多大的影响很多时候是取决于 Checkpoint
  的间隔，间隔越大，需要 replay
  的数据就会越多，从而造成中断的影响也就会越大。但是目前 Checkpoint
  间隔受制于持久化操作的时间，所以没办法做的很快。

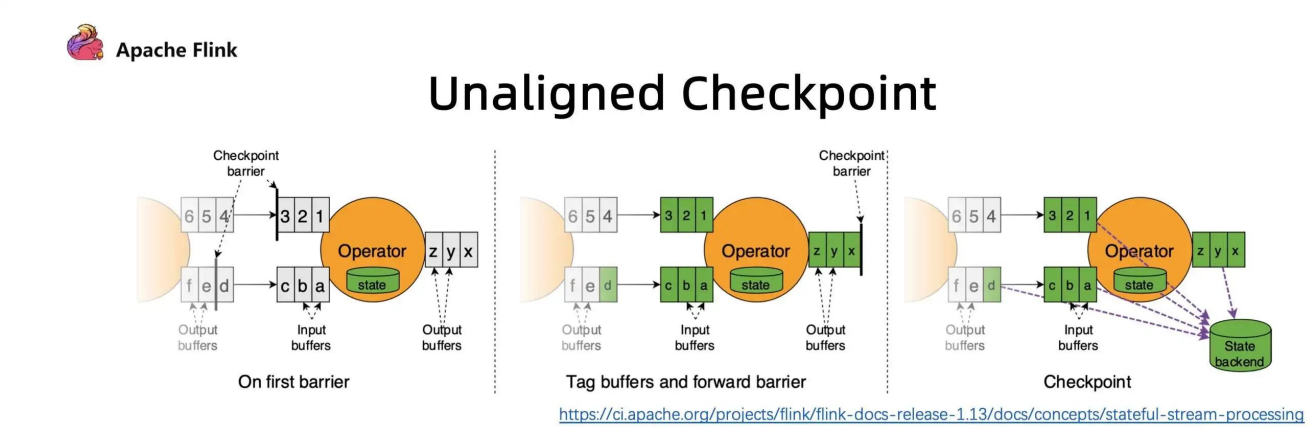
Unaligned Checkpoint

针对这些痛点，Flink 在最近几个版本一直在持续的优化，Unaligned Checkpoint
就是其中一个机制。barrier 算子在到达 input buffer 最前面的时候，就会开始触发 Checkpoint
操作。它会立刻把 barrier 传到算子的 OutPut Buffer
的最前面，相当于它会立刻被下游的算子所读取到。通过这种方式可以使得 barrier
不受到数据阻塞，解决反压时候无法进行 Checkpoint 的问题。

当我们把 barrier 发下去后，需要做一个短暂的暂停，暂停的时候会把算子的 State 和 input

output buffer
中的数据进行一个标记，以方便后续随时准备上传。对于多路情况会一直等到另外一路 barrier
到达之前数据，全部进行标注。

通过这种方式整个在做 Checkpoint 的时候，也不需要对 barrier
进行对齐，唯一需要做的停顿就是在整个过程中对所有 buffer 和 state
标注。这种方式可以很好的解决反压时无法做出 Checkpoint ，和 Barrier
对齐阻塞数据影响性能处理的问题。

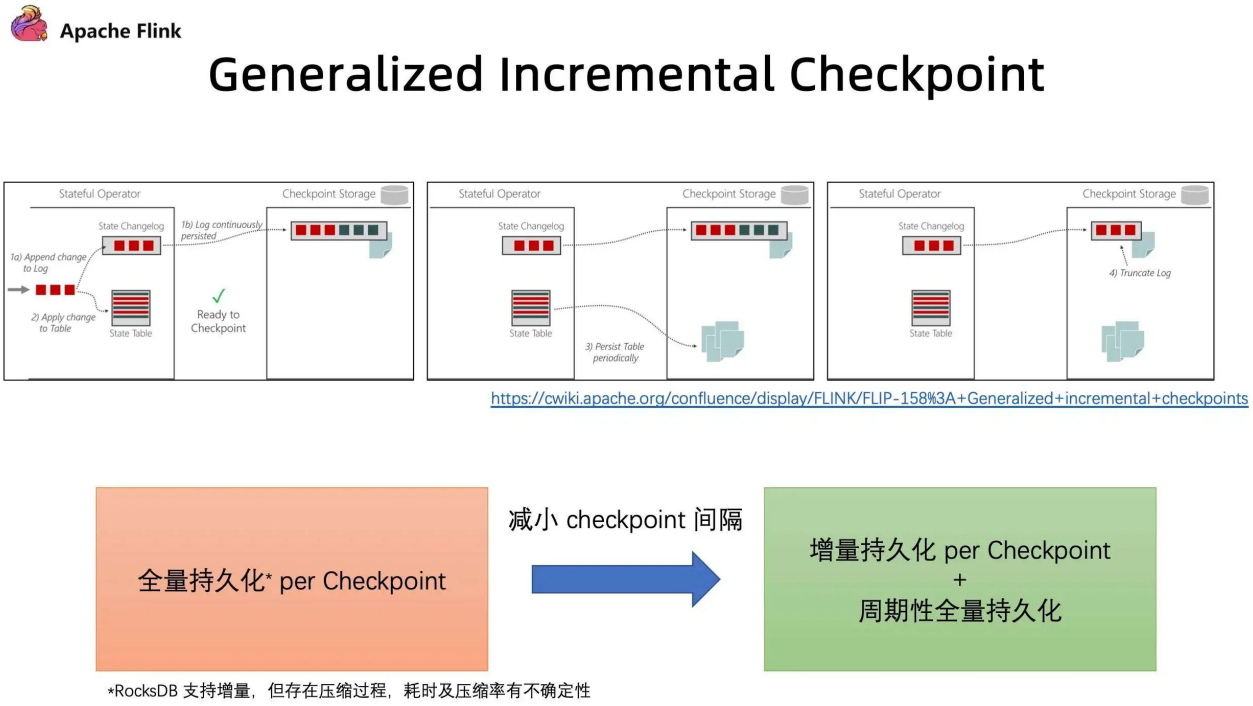


- Barrier 不受数据阻塞，解决反压时无法作出 checkpoint 的问题
- Barrier 无需对齐

如果想及时了
解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

Generalized Incremental Checkpoint

Generalized Incremental Checkpoint 主要是用于减少 Checkpoint 间隔，如左图 1 所示，在
Incremental Checkpoint 当中，先让算子写入 state 的
changelog。写完后才把变化真正的数据写入到 StateTable 上。state 的 changelog
不断向外部进行持久的存储化。在这个过程中我们其实无需等待整个 StateTable
去做一个持久化操作，我们只需要保证对应的 Checkpoint 这一部分的 changelog
能够持久化完成，就可以开始做下一次 Checkpoint。StateTable
是以一个周期性的方式，独立的去对外做持续化的一个过程。



如果想及时了
解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

这两个过程进行拆分后，就有了从之前的需要做全量持久化 (Per Checkpoint) 变成 增量持久化 (Per Checkpoint) + 后台周期性全量持久化，从而达到同样容错的效果。在这个过程中，每一次 Checkpoint 需要做持久化的数据量减少了，从而使得做 Checkpoint 的间隔能够大幅度减少。

其实在 RocksDB 也是能支持 Incremental Checkpoint 。但是有两个问题：

- 第一个问题是 RocksDB 的 Incremental Checkpoint 是依赖它自己本身的一些实现，当中会存在一些数据压缩，压缩所消耗的时间以及压缩效果具有不确定性，这个是和数据是相关的；
- 第二个问题是只能针对特定的 StateBackend 来使用，目前在做的 Generalized Incremental Checkpoint 实际上能够保证的是，它与 StateBackend 是无关的，从运行时的机制来保证了一个比较稳定、更小的 Checkpoint 间隔。

Unaligned Checkpoint 在 Flink 1.13 就已经发布了，在 1.14 版本主要是针对 bug 的修复和补充，针对 Generalized Incremental Checkpoint，目前社区还在做最后的冲刺，比较有希望在 1.14 中和大家见面。[2]

[2] Generalized Incremental Checkpoint 最终在 1.14 中没有完成。

四、性能与效率

大规模作业调度的优化

- 构建 Pipeline Region 的性能提升：所有由 pipline 边所连接构成的子图 。在 Flink 任务调度中需要通过识别 Pipeline Region 来保证由同一个 Pipline 边所连接的任务能够同时进行调度。否则有可能上游的任务开始调度，但是下游的任务并没有运行。从而导致上游运行完的数据无法给下游的节点进行消费，可能会造成死锁的情况
- 任务部署阶段：每个任务都要从哪些上游读取数据，这些信息会生成 Result Partition Deployment Descriptor。

这两个构建过程在之前的版本都有 O (n^2) 的时间复杂度，主要问题需要对于每个下游节点去遍历每一个上游节点的情况。例如去遍历每一个上游是不是一个 Pipeline 边连接的关系，或者去遍历它的每一个上游生成对应的 Result Partition 信息。

目前通过引入 group 概念，假设已知上下游 2 个任务的连接方式是 all-to-all，那相当于把所有 Pipeline Region 信息或者 Result Partition 信息以 Group 的形式进行组合，这样只需知道下游对应的是上游的哪一个 group，就可以把一个 O (n^2) 的复杂度优化到了 O (n)。我们用 wordcount 任务做了一下测试，对比优化前后的性能：

Apache Flink

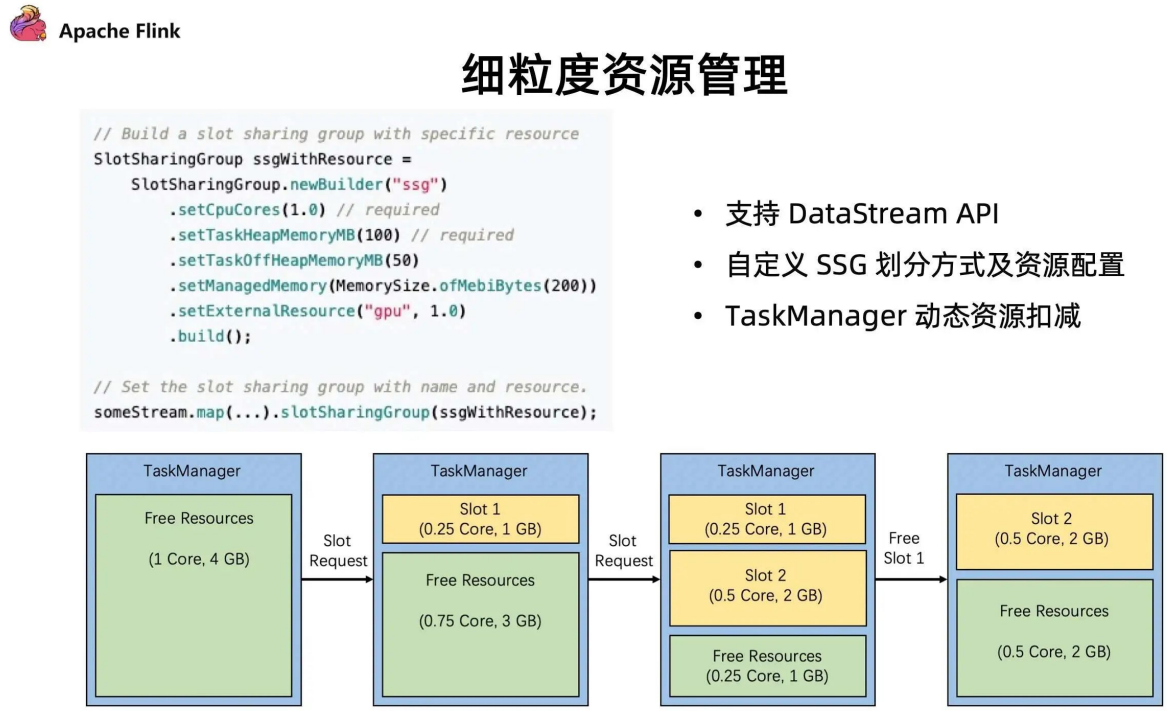
调度性能优化

	执行模式	并发度	优化前	优化后
构建 Pipeline Region	流	8k x 8k	3s 441ms	22ms
		16k x 16k	14s 319ms	107ms
	批	8k x 8k	8s 941ms	124ms
		16k x 16k	34s 484ms	308ms
任务部署	流	8k x 8k	32s 611ms	6s 480ms
		16k x 16k	129s 408ms	19s 051ms

从表格中可以看到构建速度具有大幅度提升，构建 Pipeline Region 的性能从秒级提升至毫秒级别。任务部署我们是从第一个任务开始部署到所有任务开始运行的状态，这边只统计了流，因为批需要上游结束后才能结束调度。从整体时间来看，整个任务初始化，调度以及部署的阶段，大概能够减少分钟级的时间消耗。

细粒度资源管理

细粒度资源管理在过去很多的版本都一直在做，在 Flink1.14 终于可以把这一部分 API 开放出来在 DataSteam 提供给用户使用了。用户可以在 DataStream 中自定义 SlotSharingGroup 的划分情况，如下图所示的方式去定义 Slot 的资源划分，实现了支持 DataStream API，自定义 SSG 划分方式以及资源配置 TaskManager 动态资源扣减。



对于每一个 Slot 可以通过比较细粒度的配置，我们在 Runtime 上会自动根据用户资源配置进行动态的资源切割。

这样做的好处是不会像之前那样有固定资源的 Slot，而是做资源的动态扣减，通过这样的方式希望能够达到更加精细的资源管理和资源的使用率。

五、Table / SQL / Python API

Table API / SQL

Window Table-Valued Function 支持更多算子与窗口类型 ，可以看如下表格的对比：

Apache Flink

Table API / SQL

- Window Table-Valued Function 支持更多算子与窗口类型

	Tumble	Hop	Cumulate	Session
Aggregate	1.13	1.13	1.13	1.14
TopN	1.13	1.13	1.13	
Join	1.14	1.14	1.14	
Deduplicate	1.14	1.14	1.14	

从表格中可以看出对于原有的三个窗口类型进行加强，同时新增 Session 窗口类型，目前支持 Aggregate 的操作。

1.1 支持声明式注册 Source/Sink

- Table API 支持使用声明式的方式注册 Source / Sink 功能对齐 SQL DDL；
- 同时支持 FLIP-27 新的 Source 接口；
- new Source 替代旧的 connect() 接口。

如果想及时了
解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

■ 1.2 全新代码生成器

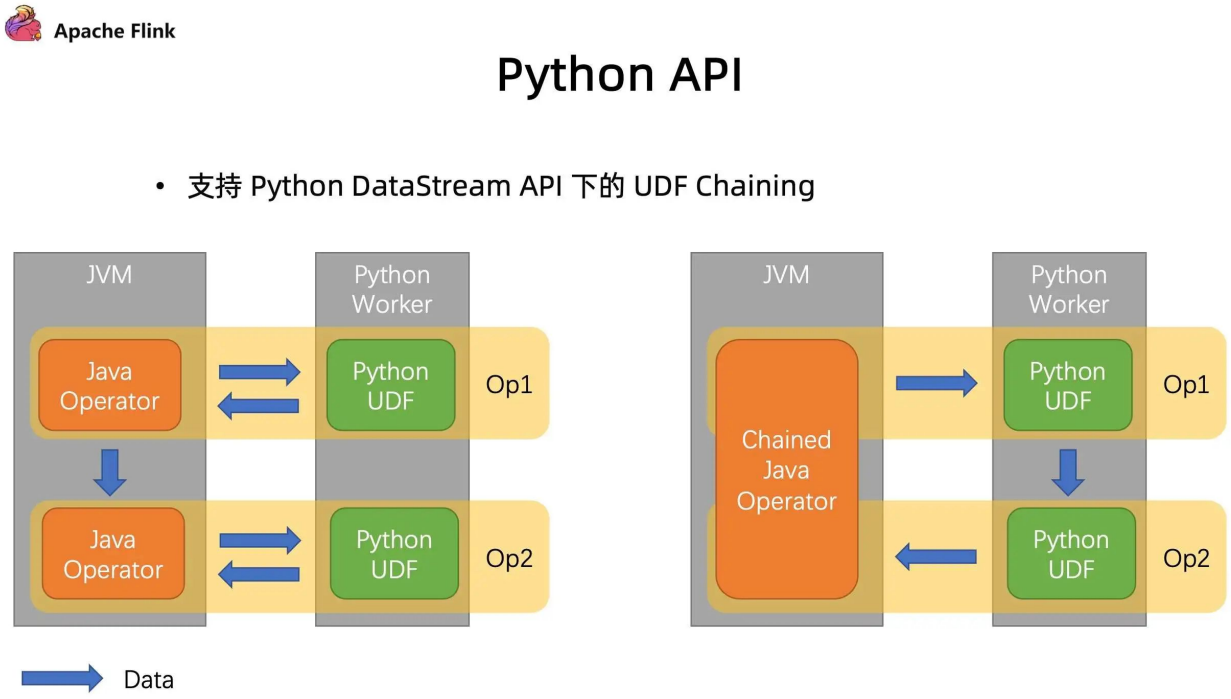
解决了大家在生成代码超过 Java
最长代码限制，新的代码生成器会对代码进行拆解，彻底解决代码超长的问题。

■ 1.3 移除 Flink Planner

新版本中，Blink Planner 将成为 Flink Planner 的唯一实现。

Python API

在之前的版本中，如果有先后执行的两个 UDF，它的执行过程如下图左方。在 JVM 上面有 Java 的 Operator，先把数据发给 Python 下面的 UDF 去执行，执行后又发回给
Java，然后传送给下游的 Operator，最后再进行一次 Python
的这种跨进程的传输去处理，会导致存在很多次冗余的数据传输。

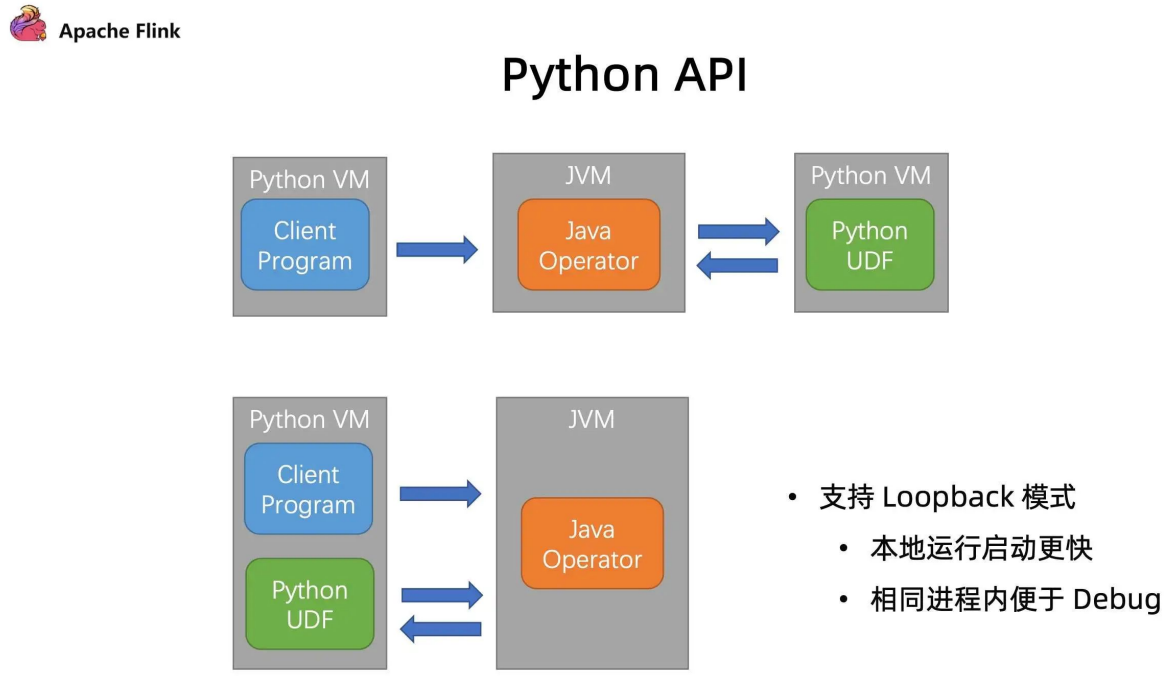


如果想及时了
解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

在 1.14 版本中，改进如右图，可以把它们连接在一起，只需要一个来回的 Java 和 Python 进行数据通信，通过减少传输数据次数就能够达到比较好的性能上的提升。

支持 LoopBack 模式

在以往本地执行实际是在 Python 的进程中去运行客户端程序，提交 Java
进程启动一个迷你集群去执行 Java 部分代码。Java
部分代码也会和生产环境部分的一样，去启动一个新的 Python 进程去执行对应的 Python UDF，从图下可以看出新的进程其实在本地调试中是没有必要存在的。



如果想及时了
解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

所以支持 lookback 模式后可以让 Java 的 opt 直接把 UDF 运行在之前 Python client
所运行的相同的进程内，通过这种方式：

- 首先是避免了启动额外进程所带来的开销；
- 最重要的是在本地调试中，我们可以在同一个进程内能够更好利用一些工具进行 debug，这个是对开发者体验上的一个提升。

六、总结

本文主要讲解了 Flink1.14 的主要新特性介绍：

- 首先介绍了目前社区在批流一体上的工作，通过介绍批流不同的执行模式和 JM 节点任务触发的优化改进更好的去兼容批作业；
- 然后通过分析现有的 Checkpoint 机制痛点，在新版本中如何改进，以及在大规模作业调度优化和细粒度的资源管理上面如何做到对性能优化；
- 最后介绍了 TableSQL API 和 Pyhton上相关的性能优化。

欢迎继续关注发版的一些最新动态以及我们在后续的 Release
过程中的一些其他技术分享和专题。

本文原文：[Flink 1.14 新特性预览]