DOI: 10.3785/j.issn.1008-973X.2022.02.010

面向 Flink 流处理框架的主动备份容错优化

刘广轩[1,2,3], 黄山[1,2,3], 胡佳丽[1,2,3], 段晓东[1,2,3]
(1. 大连民族大学 计算机科学与工程学院, 辽宁 大连 116600; 2. 大数据应用技术国家民委重点实验室, 辽宁 大连 116600;
3. 大连市民族文化数字技术重点实验室, 辽宁 大连 116600)

摘 要: 针对 Flink 任务出现故障后因为全局卷回使流处理作业恢复效率低的问题, 提出基于缓存队列的容错策略. 在作业中找出恢复时间最长的算子作为关键算子, 将其处理过的数据存储到缓存队列中, 并为其进行主动备份, 备份算子同时接受来自上游的数据以达到在故障后作业可以瞬时恢复的效果. 为了解决主动备份带来的额外消耗, 提出数据过滤算法, 备份算子在每次处理数据前会到缓存组件中检索当前数据, 以判断是否继续处理. 当 Flink 算子自身出现故障后, 利用策略中的缓存队列与 Flink 的 JobManager 将故障发生时的数据信息发送给备份算子, 在备份算子接收到数据后, 实现即时恢复的效果. 利用 4 项评价指标对策略进行评估, 结果表明, 与 Flink1.8 的故障恢复模式相比, 所提策略在 Flink 任务故障恢复速度上有显著提升, 当故障次数分别为 1、2、3、4 时, 恢复效率分别提高 56.3%、51.3%、46.2% 和 45.8%; 而在处理时延、CPU 利用率以及内存使用率方面仅产生极小的代价.

关键词: Apache Flink; 流处理容错; 主动备份; 故障恢复; 缓存队列
中图分类号: TP 316.4 **文献标志码:** A **文章编号:** 1008−973X(2022)02−0297−09

Fault tolerant optimization of active backup for Flink stream processing framework

LIU Guang-xuan[1,2,3], HUANG Shan[1,2,3], HU Jia-li[1,2,3], DUAN Xiao-dong[1,2,3]
(1. *College of Computer Science and Technology*, *Dalian Minzu University*, *Dalian* 116600, *China*;
2. *State Ethnic Affairs Commission Key Laboratory of Big Data Applied Technology*, *Dalian* 116600, *China*;
3. *Dalian Key Laboratory of Digital Technology for National Culture* , *Dalian* 116600, *China*)

Abstract: A fault-tolerant strategy based on cache queue was proposed, aiming at the problem of low efficiency of stream processing job recovery due to global rollback after Flink task fails. In the job, the operator with the longest recovery time is taken as the key operator, the processed data are stored in the buffer queue, and active backup is performed for it. The backup operator will also accept the data from the upstream to reach the the effect that the job can be restored instantaneously after a failure. In order to solve the additional consumption caused by active backup, a data filtering algorithm was proposed. The backup operator will retrieve the current data from the cache component before processing the data each time to determine whether to continue processing. When the Flink operator itself fails, it will use the buffer queue in the strategy and Flink's JobManager to send the data information at the time of the failure to the backup operator. When the backup operator receives the data, it will realize the effect of instant recovery. The strategy was evaluated on four evaluation indicators. Compared with the failure recovery mode of Flink1.8, the proposed strategy had a significant improvement in Flink task failure recovery. The recovery efficiency was increased by 56.3%, 51.3%, 46.2% and 45.8% under failure times of 1, 2, 3 and 4 separately. And the proposed strategy brings only a very small price in terms of processing delay, CPU utilization and memory usage.

Key words: Apache Flink; stream processing fault tolerance; active backup; failure recovery; buffer queue

收稿日期: 2021−07−18. **网址:** www.zjujournals.com/eng/article/2022/1008-973X/202202010.shtml
基金项目: 国家重点研发计划云计算和大数据重点专项(2018YFB1004402).

作者简介: 刘广轩(1997—), 男, 硕士生, 从事大数据和流计算研究. orcid.org/0000-0003-3773-7594. E-mail: liuguangxuan_ustl@163.com
通信联系人: 黄山, 男,讲师. orcid.org/0000-0003-1758-755X. E-mail: huangshan@dlnu.edu.cn

大数据实时处理引擎已经从 Hadoop 承载的 MapReduce[1] 发展到以内存计算为主的 Spark[2] 再到现如今的有状态流处理引擎 Flink[3]. Flink 的出现促进了上层应用的快速发展并实现了更进一步的实时性，其纯流处理的机制有效避免了批处理事件以时间为划分后将中间结果带到下次批运算的情况，Flink 更符合实时数据的状态特质.

在实时计算场景中，很多任务会直接服务于线上，其输出时延和稳定性会直接影响线上产品的用户体验. Flink 的计算效率较高，具有高吞吐低延迟的特性，在处理性能方面，Flink 相比于 Spark Streaming 具有更高的处理效率，并且在没有网络延迟的情况下，Flink 的处理效率是 Storm 的 37.5 倍[4]. Flink 在容错方面有基于 Chandy-Lamport[5] 算法的分布式快照策略为支撑，目前已经应用在阿里巴巴、字节跳动、美团等大型互联网公司的实际场景中.

然而，Flink 现有的架构设计使得在复杂拓扑下单个 Task 失败就能使所有 Task 重新部署，耗时可能会持续几分钟，导致作业的输出断流，降低了业务处理的实时性与稳定性. 本研究针对这一问题开展研究，提出基于缓存队列的主动备份容错模型，然后提出缓存队列数据过滤算法，优化了主动备份处理效率. 最后在 Flink 上实现了本研究提出的优化策略，并通过实验验证容错优化机制的有效性.

1 相关研究

现有的流处理引擎容错策略分为 2 大类，一类是主动备份，一类是被动备份.

主动备份策略是指对任务的一种热备份，在一些流处理系统中较为常见且实现过程较简单. 主节点和从节点之间通过网络通信，在正常情况下，主机处于工作状态，从机处于等待状态，两者通过监听心跳感知状态变化，当发现主节点出现异常，从节点立刻代替主节点，完成主节点的任务. 主动备份策略可以使得备份任务在出现故障后实现实时恢复，故障的出现基本不会影响整个应用，保障了系统的稳定性. 然而，这种策略会使得任务在备份时计算多次，使得处理效率降低. 关于主动备份的优化，Balazinska 等[6] 提出延迟、处理和纠正 (delay，process，and correct, DPC) 模型，是基于主动备份的流处理容错方法，在保证了低处理延迟的基础上，容忍多个同时发生的故障以及恢复期间发生的任何进一步故障，同时还保证了最终结果的一致性. Hwang 等[7] 提出在主动备份时管理备份副本的算法，系统中的任何副本都可以使用上游副本中最先到达的数据，该系统实现了低延迟处理以并保证了服务器和网络问题的鲁棒性.

被动备份[8] 的主要思想是将每个任务的自身最新状态定期备份到内存或外部存储介质中，当发生故障后，应用可以在该介质处进行任务的恢复. Flink 的基于检查点机制的容错方法是一种被动备份的思想，检查点机制在大多数情况下可以提高系统的效率，而且在生产环境中已经被广泛应用，但是该机制仍存在以下问题：一是检查点频率对系统处理效率以及中央处理器 (central processing unit, CPU) 使用的影响，二是从检查点处恢复会带来一定的卷回开销，造成系统资源的浪费以及处理效率的低下. 针对以上问题，较多文献以检查点为突破口进行相关的优化，其中 Chandramouli 等 [9] 通过实验证明 Flink 检查点间隔越大，带来的处理延迟越小并且 CPU 的使用率越低，但与此同时故障恢复会带来一定的损耗.

现如今大多数处理引擎将优化目标放在了检查点上面. Benoit 等[10] 对一些无法立刻检测到的潜在错误提出平衡算法，使得检查点和其验证达到最佳平衡状态. Akber 等[11] 根据任务的失败率决定检查点间隔以优化应用的处理效率. Zhuang 等[12] 提出流处理系统最优检查点模型，通过该模型可以求解出左右检查点间隔参数. Lombardi 等[13] 提出了分散的因果检查点容错技术，在不影响恢复效率的情况下显著降低了基于检查点的方法运行时的开销. 刘智亮[14] 提出面向流式应用负载动态变化，支持在线自适应调节间隔的检查点机制. 郭文鹏等[15] 等结合 Flink 系统的迭代数据流模型，进一步提出基于头尾检查点的悲观迭代容错机制，该容错机制以非阻塞的方式编写检查点，充分结合 Flink 迭代数据流的特点，将可变数据集的检查点注入迭代流本身. 通过设计迭代感知，简化系统架构，降低检查点成本和故障恢复时间. 目前生产环境中使用的流处理系统所采用的容错策略大多亦是基于传统流计算容错方法的混合与改进. Apache Storm 通过周期性地发送 Checkpoint 消息，让所有算子进行状态备份. 并且，在元组级别，使用 acker 线程追踪由源头发送的元组信息组

成的元组树及确认消息，通过比对这两方面信息，保证元组不会丢失. 失败的元组会由源头进行重放重新处理. Spark Streaming[16] 采取离散化的编程模型，将流应用映射成一系列的微批处理. 它定期将算子的状态和数据信息保存在弹性分布式数据集 (resilient distributed datasets, RDD) 中. 如果 RDD 中某个分区发生故障，可以通过其他机器并行计算从而加速恢复. Apache Flink[17] 基于栅栏模型实现了轻量级的异步检查点机制，结合支持稳定存储的数据源可以保证元组被系统正确处理，是检查点结合上游备份的容错策略.

不论是基于传统的主动备份、被动备份、上游备份策略，还是基于检查点的卷回恢复机制，单一、静态的容错机制难以满足不同应用场景的实际需求.

2　问题归纳

2.1　Flink 的容错策略介绍

实时大数据处理系统须提供容错机制，以保证系统可以应对如进程失败、节点宕机或网络中断等故障. Flink 将故障分为 Task failover 与 Master failover 两大类分别处理.

Master failover 指的是 JobManager 出现故障，此时 TaskManager 可以正常运行. 针对这种情况，Flink 须将所有任务重新执行.

Task Failover 指的是单个 Task 由于某种原因 (如用户程序逻辑存在逻辑上的错误、Task 自动退出或者 TaskManager 因为系统环境的问题出错退出) 而执行失败的故障. 表现为 Master 还在，但是某一个 Task 失联，最终通过下游的 Task 抛出异常. 针对这种情况，Flink 提供了 3 种恢复模式. 1) Restart-all 恢复模式. 该恢复模式是基于 Flink 提供的一种基于检查点 (Checkpoint) 的故障恢复机制实现的，该机制存在仅一次 (exactly-once) 和至少一次 (at-least-once) 这 2 种语义下做快照的方式，使 Task 只须从上一个 Checkpoint 处进行恢复重新运行即可. 2) Restart-individual 恢复模式. 在该模式下，某 Task 出现故障后，仅须重启该 Task 即可，然而此模式仅适用于各个 Task 之间没有数据传输的情况，应用范围较小，本研究不作重点讨论. 3) Restart-Region 模式. 该模式将 Flink 批处理作业图中 Task 以 Pipeline 方式传输的子图称为一个 Region，从而将整个 Flink 执行图分为多个 Region，那么出现故障后，只须重启该 Region 部分的 Task 即可. 该模式仅适用于批处理作业，不是本研究重点.

2.2　Flink 容错策略存在的问题

Restart-all 基于 Checkpoint 的全局卷回恢复的形式来重启整个任务，这种策略会带来一定的恢复代价与资源的浪费. 任务重启需要大量的时间，并且任务在恢复时只能从上一个 Checkpoint 处进行恢复，即存在任务重复计算的情况. 假设现有某 Flink Job 的 StreamGraph 如图 1 所示. 图中，T_1、T_2、T_3、T_4 为各个算子之间传输的平均时间. 若在某一时刻算子 O3 所在节点出现故障导致 Flink Job 从上一次 Checkpoint 处恢复，此时任务恢复到故障之前的所需时间为

$$T_{\mathrm{rc}} = T_{\mathrm{rp}} + T_{\mathrm{ol}}. \tag{1}$$

式中：T_{rc} 表示任务在出现故障后的总的恢复时间；T_{rp} 为任务重播时间，即所有算子状态恢复到上一个 Checkpoint 保存的状态的时间；T_{ol} 为任务重载时间，即故障发生到节点重启到正常状态的时间，该部分耗费时间也是最长的. Flink 一次成功的 Checkpoint 包含了所有任务在某一时间点的状态，在该时间点中所有任务都恰好处理完一个相同数据，即图中所有 State 存储的都是与同一份数据相关的中间结果，从 Source 端开始处理到 Sink 端结束并将结果存储到 Checkpoint Source State 中，这算一次成功的 Checkpoint. 若 O3 出现故障，当前 Checkpoint 保存的状态是无法算作一次完整的 Checkpoint 的，因此只能从上一次成功的 Checkpoint 处取数据进行恢复，重播所需的时间为

$$T_{\mathrm{rp}} = n(T_1 + T_2 + T_3) + (n-1)T_4;\ n \geqslant 2. \tag{2}$$

式中：n 为故障位置与上一次成功的 Checkpoint 保存的偏移量相差的轮数. 由于 Checkpoint 存储在内存或外部存储引擎中，制作 Checkpoint 时将会有输入/输出 (input/output, I/O) 以及 CPU 消耗，所以 Checkpoint 的间隔也会对 Flink 的处理时延以

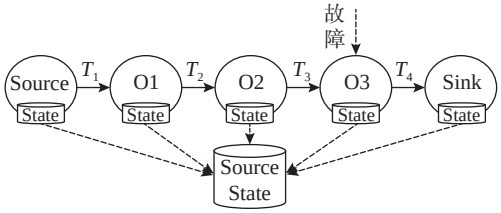

图 1　Flink Job 的 StreamGraph

Fig.1　StreamGraph of Flink Job

及故障恢复的时间产生影响. Checkpoint 设置的间隔越小就会给 Flink 带来越高的处理时延，而间隔越大在任务出错后故障恢复的时间也会越大. 因此，Checkpoint 的间隔应针对任务的出发点不同进行调整.

2.3 优化目标

一些应用 (如金融系统、实时计算、网络监控) 要求系统在规定的时间内完成数据分析的任务，因此在这些实时计算任务中，除了系统自身故障、网络波动和处理逻辑之外，计算超时也会被判定为整体作业的失败，即发生故障. 如表 1 所示为数据清洗实时处理系统的日志信息，该系统要求任务在每日的 8:00 准时接受前端数据，并通过 Flink 进行数据处理，要求必须在 8:10 之前完成任务并发送给下一个组件处理.

由表 1 可以发现，流处理框架，如 Flink，在进行实时处理面对任务故障问题时表现出不够实时性，常常须消耗一定的时间去进行故障恢复，从而导致任务失败 (如表中日期 1.10 和日期 1.11)，导致整体效率变低，造成任务失败. 同时，底层处理组件 keyGroup 的数目限制 (必须是 2 的幂次) 导致 Flink 任务存在木桶效应，整体任务受到短板节点的影响，会出现如表 2 所示的，即使扩大并行度 (测试案例为并行度从 8 扩大到 12) 也不会提升甚至会降低任务吞吐量的情况. 表中，P 表示总的并行度，K_1 表示分配了 1 个 KeyGruop 的并行度数量，K_2 表示分配了 2 个 KeyGroup 的并行度数量，T 表示任务每分钟的吞吐量. 当并行度为 12 时，将有 4 个 subtask 去处理 2 个 KeyGroup 数据，将会造成整个任务的瓶颈. 在有一定额外空闲资源的情况下，由于木桶效应的存在，Flink 对资源的合理利用要求较高，若利用不好或者资源没有达到一定量的需求，整体效率无法提升，即存在对空闲资源利用不合理的情况.

根据前文概述，相对其他流处理系统，Flink 的处理效率较高. 但是，Flink 难以提供高效的故障恢复机制，主要原因是当某节点出现故障后，在该节点恢复时必须进行全局重启，重载与重播带来的巨大时间消耗对于一些线上业务来说是难以接受的. 同时，底层组件的数目限制导致 Flink 常常出现耗费资源却无法提升效率的情况，因此结合上述问题，本研究针对流处理系统提出的优化目标如下：1) 当系统中某一个任务出现故障时，无须全局重启任务，且故障节点能瞬时恢复；2) 非故障 Task 正常为线上提供服务；3) 故障恢复后不影响整体任务的最终结果，即不会在失败的组件中丢失任何信息，且能保证数据全部被处理且仅处理一次；4) 充分利用空闲资源去解决上述目标.

表 1 Flink 任务日志信息

Tab.1 Log information of Flink task

日期	开始时间	故障次数	完成时间	完成
1.7	8:00	0	8:08:20	是
1.8	8:00	1	8:09:10	是
1.9	8:00	2	8:10:02	否
1.10	8:00	3	8:10:23	否
1.11	8:00	1	8:09:08	是

表 2 增加并行度对吞吐量的影响

Tab.2 Impact of increasing degree of parallelism on throughput

P	K_1	K_2	T
8	8	0	2000
12	8	4	1923
16	16	0	4335

3 基于缓存队列的容错策略

3.1 缓存队列容错策略介绍与实现

缓存队列的容错策略 (buffer queue backup strategy，BQBS) 采纳主动备份的快速恢复的思想，同时对主动备份带来的重复计算的问题进行优化. 本研究提出的 BQBS 策略将会大幅提升 Flink 任务在出现故障后的恢复效率. 该策略以 Flink Job 的某个 task 为基础，流程如图 2 所示.

1）在数据流准备进入须备份的 Task 之前须进行预处理操作，为数据流中的每条数据元组添加唯一标识 ID. 2）上游算子利用空闲资源将数据流传输到主算子以及其备份算子中，并保证备份算子与主算子同时接受到相同的数据. 3）在 Task 与其备份算子 Task1 之间设置内存级别的 Buffer Queue 组件，其最大优势是可以快速读写数据，并在该 Queue 中缓存 Task 已经处理或正在处理元组的 ID 以及对应的数据信息. 4）在 Task 出现故障后，JobManager 立刻感知 Task 心跳超时，产生故障信息，将发生故障的元组 ID 发送给备份算子. 5）备份算子在接受到故障信息以及元组 ID 后，

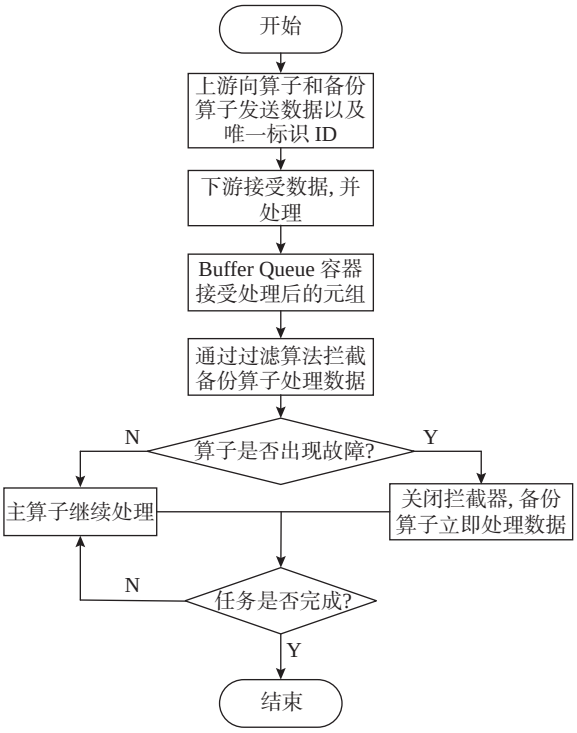

图 2　缓存队列容错策略工作流程

Fig.2　Workflow of Buffer Queue backup strategy

重新在 Buffer Queue 中检索并拾起该 ID 对应的数据, 激活备份算子与下游算子之间的数据传输通道, 该算子此时可以做相应的逻辑处理, 主从节点身份进行互换.

步骤 1) 对上游的中间结果添加一个唯一表示 ID, 该 ID 在故障恢复中相当于一个索引的作用, 可以使备份算子立刻查找到故障数据. 在步骤 2) 中, 上游算子的数据利用空闲资源进行并行传输, 即 T 主算子与备份算子会同时接受相同数据并且处理逻辑完全相同. 为了保证不同时出现故障, 两者会运行在不同的节点中. 在步骤 3) 中 Buffer Queue 在对 Flink 任务容错性能提升的前提下还可以通过其自身的缓存与过滤重复元组的功能保证任务数据处理仅一次的语义. 其中过滤算法如下所示.

算法 1　Buffer Queue 的过滤算法

输入：任务拓扑, 添加 Tuple ID 的流数据, 最大缓存值 MAX_MEMORY.

输出：无.

```
1. if Buffer Queue.containskey (TupleID)
2.  ACTION="FILTER"
3. else
4.  Buffer Queue.EnQueue (TupleID)
5.  ACTION="ADD"
6. end if
7. if operator.ACTION="ADD"
8.  operator 做逻辑运算
9. else if
10.  operator.ACTION="FILTER"
11.  operator 不做逻辑运算
12. End if
13. If Buffer Queue.Memory≥MAX_MEMORY
14.  Buffer Queue.ClearQueue()
15. end if
16. end
```

根据算法 1, Buffer Queue 作为关键组件, 会按顺序缓存处理过的数据及其 ID, 若 Buffer Queue 中没有当前数据的 ID, Buffer Queue 会将对应的数据信息以及其 ID 加入队列中, 发出“ACTION=ADD”的信号; 若 Buffer Queue 中存在该 TupleID, 则会发出“ACTION=FILTER”信号. 当前主算子和备份算子在做运算之前时, 都会对 ACTION 进行取值验证, 当 ACTION 的值为“ADD”时, 主算子将会执行相应的逻辑, 当接受到“FILTER”时, 主算子不会做处理, 也就不会存在重复计算导致的高资源消耗问题, 也可以保证结果的准确性. 当主算子出现故障时, 主算子与下游失去通信, 不会接受到新元组, 而备份算子始终接受下游算子的数据信息. 按照算法 1, 备份算子接受到的 ACTION 将会是“ADD”, 因此会做相应处理. Buffer Queue 的大小并不是无限制的, 会受到最大容量 MAX_MEMORY 的限制, 当超出该上限时, 则执行队列的 FIFO 规则, 新的元组将替换旧的元组. Flink 使用算法 1 可以提升上游传输故障恢复效率, 如图 3 所示, 当上游算子 A 处理完数据后, 信息同时发送给主算子 B 和备份算子 B1, Buffer Queue 中缓存了主算子 B 已经处理或正在处理的数据 ID. 当备份算子 B1 准备处理数据时, 会在 Buffer Queue 中检测, 如果存在该 TupleID, 则对备

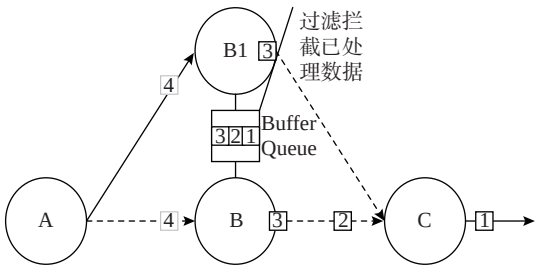

图 3　正常情况下缓存队列的拦截作用

Fig.3　Interception function of Buffer Queue under normal circumstances

份算子要处理的数据进行拦截，备份算子就不会继续处理该数据.

如图 4 所示，当上游算子 A 与 B 的数据传输出现故障后，下游算子 A 不再向 B 发送数据，A 与备份算子 B1 的传输通信保持正常，此时 Buffer Queue 中并没有 B1 中的数据信息，因此就会缓存来自备份算子 B1 中的数据，并开启拦截器，B1 与下游算子的 channel 被激活，可以达到瞬时恢复的效果.

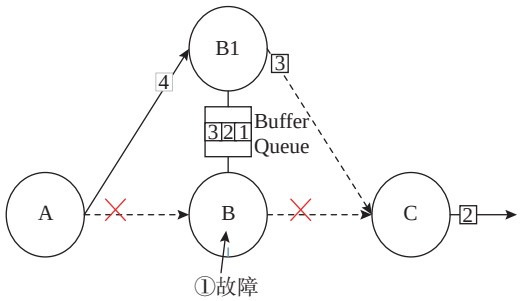

图 4 备份算子处理数据示意图

Fig.4 Schematic diagram of backup operator processing data

当算子自身出现故障后，根据本研究提出的算子自身故障恢复算法进行处理，具体步骤如下所示.

算法 2 算子自身故障恢复算法

输入：任务拓扑，operator B 注入进程级别的故障，operator B 的上游 operator A，算子 B 的下游 operator C，算子 B 的备份算子 B1

输出：无

1. operator A 与 operator B 的 Channel 关闭
2. operator B 向 operator C 发送 Exception
3. operator C 向 JobManager 发送当 Cache 中最大的 TupleID
4. JobManager 向备份算子 operator B1 发送接收到的 TupleID 并且设置 operator B1 的 ACTION="ADD"
5. TupleID=TupleID+1
6. 在 Buffer Queue 中查找 TupleID 对应的算子信息
7. y = 当前 Buffer Queue 中最大的元组 ID
8. for i=TupleID to y
9. operator B1 做对应 TupleID 的逻辑运算
10. End for

如图 5 所示，当主算子 B 出现故障后，JobManager 会通过心跳超时感知程序异常，与此同时，不论是逻辑错误还是进程级别的故障，Flink 本身算子的上下游都会感知到连接状态的变化，

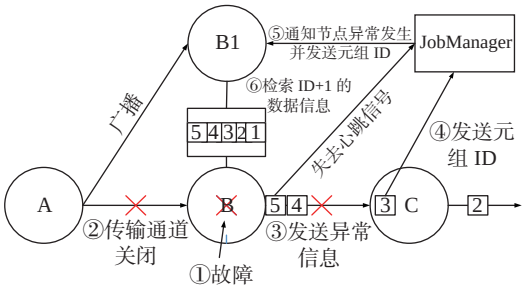

图 5 算子自身故障恢复过程

Fig.5 Recovery process of operator's own fault

即算子的上下游都会感知到故障的发生. 此时，算子的下游 C 会从当前算子 Cache 中找出最大的元组 ID 发送给 JobManager，因为进入到算子 C 的数据都是被算子 B 成功处理后的数据，同时 Flink 中每个算子都有一个 Cache 用来缓存待处理算子，若从算子 C 的 Cache 中取出最大的元组 ID，该 ID 就是算子 B 在发生故障前最后一个正确处理的算子. 同时为了让备份算子感知错误，在主算子出现故障后，JobManager 会收集当前出现故障数据的元组 ID，并将 ID 发送给备份算子，当备份算子接受到 ID 后，会从 Buffer Queue 中检索 ID+1 对应的数据信息. 选择元组 ID+1 的原因是，当前 ID 对应的数据元组已经被主算子处理，如图中 ID=3 的算子此时已经进入下游算子 C，程序在运行到元组 ID=4 的算子时发生故障，因此，当备份算子接受到算子 ID 后，须在 Buffer Queue 中检索 ID+1 的元组信息. 当备份算子检索到信息后，拾起数据开始继续做处理，此时主、从算子身份互换，之前的备份算子立即切换成为主算子.

3.2 效率评估

所提出的策略关键在于解决了 Flink 无法充分利用空闲资源以及出现故障后延迟的问题. 根据前文提出的木桶效应以及式 (4) 可以得出，在一定程度上扩大 Flink 任务的并行度也无法提升整体效率，因此本研究充分利用空闲资源，将资源利用在算子备份以及 Buffer Queue 维护上.

本研究重点解决了 Flink 自身故障恢复后带来的重载延迟，即在整体恢复时延中去掉了 T_{over} 这一最影响故障恢复效率的因素，并缩短了全局卷回导致重复计算带来的延迟. 若是上游传输故障，本研究提出的模型可以达到瞬时恢复，并不影响整体效率. 但是在实际生产中，较为常见的故障是发生在算子本身，本研究策略在解决算子自身故障延迟时会带来一定的资源损耗，主算子的上游算子须将其处理的数据做 2 次分发，此处

带来的损耗记作 $L^{(up)}$, 若使用该模型运行 n 条数据, 那么带来的损耗为

$$L = nL^{(up)}. \tag{3}$$

若备份算子到下游算子传输延迟为 T_{bd}, 则此时算子整体的时延为

$$T_{re} = T_{bd} + L. \tag{4}$$

实际中, 该策略还会带来一定的网络通信消耗, 但是这部分的消耗与 Flink 原本的消耗基本相似, 因此在式(4)中不进行估计.

4 实验结果与分析

4.1 实验环境

本实验由 5 台服务器组成, 1 台主节点, 3 台从节点, 1 台应用于备份的节点. 如表 3 所示为 Flink 集群中单个节点的主要配置信息, 如表 4 所示为实验主要的软件环境以及对应的版本号.

选择问答网站的用户回答信息作为数据集, 该数据集是一个嵌套 JSON, 包含用户信息、点赞数、收藏数、评论数. 对该数据进行数据清洗, 根据点赞数来为评论添加等级信息. 首先使用 Kafka 对数据进行采集, 从 Redis 中获取到点赞与等级之间的映射关系, 再使用 Flink 对数据进行抽取、转换、加载, 在完成数据解析后, 将数据写入到 Kafka 的另一个 topic 中, 最终使用 Flume 将数据进行分类落盘存储到 HDFS 中. 本实验所有计算的并行度为 8, 每个 TaskManager 所拥有的 slot 数为 8, 根据需求使用 Flink 中的 Source、Connect、FlatMap、Filter、Map、Sink 算子进行运算.

表 3　Flink 集群硬件配置信息

Tab.3　Hardware configuration information of Flink cluster

硬件	配置信息
处理器	Intel i7-8 700 CPU @ 3.20 GHz
CPU核心数	6核
主频	3.2 GHz
内存	64 GB
硬盘	100 GB SSD

表 4　Flink 集群软件配置信息

Tab.4　Software configuration information of Flink cluster

软件名	版本号	软件名	版本号
Centos OS	7.4.1	Java	1.8.0
Flink	1.8.0	Kafka	0.10.1.0
Hadoop	2.7.5	Flume	1.7.0

4.2 评估标准

主要选择 10、50、100 万 3 个不同数据量的信息进行对比. 主要选取处理时延、吞吐量、故障恢复时间作为评估标准.

1)处理时延对比分析. 时延 (latency) 指从数据输入至计算引擎中被处理开始到最终输出的时间, 单位为毫秒. 处理时延反映了一个系统的实时性能, 延迟越低用户体验越好, 时延表达式如下:

$$T_L = T_E - T_B. \tag{5}$$

式中: T_E 为数据处理完成时间, T_B 为数据流进入系统时的时间. 本研究优化的目标就是在缩短故障恢复时延的前提下, 尽量对 Flink自身的处理时延不产生影响.

2)故障恢复时间对比分析. 故障恢复时间 (recovery time) 指前文所说的重载与重播时间, 即故障发生开始到任务恢复到上一次数据源处理的位置所需的时间.

3)CPU 利用率分析. CPU 利用率指一段时间内 CPU 的使用量, 本研究实验通过 CPU 利用率来验证计算资源占用情况.

4)内存使用率分析. 本研究通过实验对比 Flink-RestartAll 与 Flink-BQBS 在处理任务时的内存使用率.

本研究先对任务中的每一个算子进行故障影响分析, 即对每个算子人工注入进程级故障, 以推算算子的关键度, 根据算子的关键度找出相对关键的算子并根据本研究提出的算法做备份, 并进行故障恢复时间对比分析, 来验证本研究提出的算法的优化效果. 为算子注入 50 次故障后各自的平均恢复时间 T_R 如表 5 所示. Checkpoint 的间隔为 30 s, 由表 5 可以看出, 本次任务的 FlatMap 较为关键, 该算子参与大量的逻辑运算, 并处理了来自 2 条数据源的信息, 因此, 本研究按照表 5 的恢复时间从大到小的顺序进行备份实验.

在第 1 个实验中, 对 FlatMap 算子使用本研究策略, 在不同规模的数据下产生的处理时延对比如表 6 所示, 其中 Flink 的 Checkpoint 间隔设置为 30 s. 可以看出, 本研究所提算法对数据处理的延迟产生的影响较小, 尤其当数据量在 100 万以下时, 与 Flink 原来的处理延迟没有太大差别; 当数据量超过 100 万时, 使用本研究所提出的策略后

表 5 Flink 任务故障平均恢复时间

Tab.5 Average recovery time of Flink task failure

算子	T_R/s	算子	T_R/s
Source	45	Filter	67
Connet	65	Map	53
FlatMap	71	Sink	39

表 6 不同数据量下平均处理延迟对比

Tab.6 Comparison of average processing delay under different data volume values

数据量/万	处理时延/s	
	Flink-RestartAll	Flink-BQBS
10	82.0	84.5
50	170.0	176.0
100	355.0	369.0
200	723.0	739.0

处理时间会比 Flink 原本的略高，约会产生 4% 的额外时延，原因是上游算子进行 2 次分发带来的延迟会随着数据量的增大而逐渐增大.

在第 2 个实验中，分别对 Flink-RestartAll 与本研究提出的 Flink-BQBS 分 4 次对 FlatMap 所在节点随机注入进程级故障，2 个模式的任务恢复结果如表 7 所示. 在 Flink-RestartAll 模式下每发生一次故障，平均恢复时间约为 70 s，其中任务重载时间约为 30 s，根据前文公式推算，本研究提出的 BQBS 策略可以避免任务的重载，且重播时间也会缩短，当第 1 次出现故障时，平均故障恢复时间为 31 s，效果提升了 56.3%，在第 2 次出现故障后恢复时间与 Flink 基于 Checkpoint 的全局卷回策略第 1 次发生故障的恢复时间大致相同，平均为 73 s. 当第 4 次出现故障后，恢复时间约为 157 s，也远好于同等故障下 Flink-RestartAll 的累计故障恢复时间.

表 7 不同恢复方式下平均恢复时间对比

Tab.7 Comparison of average recovery time under different recovery modes

故障次数	Flink-RestartAll累计重启时间/s	Flink BQBS故障恢复时间/s
1	71	31
2	150	73
3	210	113
4	290	157

本研究所提出的 Flink-BQBS 出现的在不同故障次数下恢复时间产生差异的原因是，本研究策略中故障恢复后导致的主从节点互换会使得下一次故障后的从节点与主节点之间接受的数据偏移量有所差距，在下一次出现故障后总会在算子重播上花费一些时间，且平均时间约为 30 s.

在第 3 个实验中，分别对比 Flink 正常运行的情况下以及出现故障时系统的 CPU 利用率. 在正常情况下，本研究策略与 Flink 的 CPU 利用率基本相似，在出现故障时，本研究提出的算法会导致更高的 CPU 利用率，大致会消耗 8% 的 CPU，出现这个的原因是因为下游算子频繁的广播数据到下游的操作会导致系统花费一定的资源.

在第 4 个实验中，对比了 Flink-RestartAll 与 Flink-BQBS 的内存使用率. 在任务正常运行情况下，Flink-RestartAll 将会占用 40.01% 的内存，Flink-BQBS 会占用 42.70% 的内存，本研究提出的算法内存使用率高的原因是在算法流程中主算子接受的数据会被缓存到缓存队列中，而缓存队列是基于内存的，存储的数据与缓存队列自身都会占用系统的内存.

根据实验，得出 Flink 的 Restart-all 恢复策略与本研究提出的基于 Buffer Queue 的恢复策略的对比信息，如表 8 所示. 表中，η_n、η_f 分别为 CPU 正常、故障利用率，δ 为内存使用率.

综上，本研究策略能在占用较低资源的情况下保证算子在出现故障后迅速恢复，并且能够在该任务出现 3 次及以下故障时，比 Flink 默认的恢复策略效率更高，恢复时间更短，系统稳定性得到了进一步提升.

表 8 Flink-RestartAll 与 Flink-BQBS 对比信息

Tab.8 Comparative information of Flink RestartAll and Flink-BQBS

对比算法	恢复模式	T_R/s	η_n/%	η_f/%	δ/%
Flink-RestartAll	全局卷回	71	62.0	76	40.01
Flink-BQBS	单点恢复	31	62.5	82	42.70

5 结 语

针对 Flink 任务出现故障后因全局卷回策略造成流处理作业恢复效率低的问题，提出基于缓存队列的主动备份容错策略（BQBS），该策略主要借鉴了主动备份将任务从故障处快速恢复的思想以解决 Flink 容错效率低的问题. 该策略对主动

备份中存在的备份节点重复计算的问题做了相应的优化，结合缓存队列提出一种过滤算法，实现了对重复数据过滤的功能，该算法可以提升 Flink 在发生上游传输故障时的恢复效率. 同时为了解决 Flink 算子自身故障恢复问题，提出算子自身故障恢复算法，该算法会借鉴 Flink 的 JobManager 发出的信号以及下游算子缓存的数据信息将故障发生时的数据发送给备份算子，当备份算子接收到数据后，实现即时恢复的效果. 通过对比 Flink 的 Restart-all 恢复模式与提出的 Flink-BQBS 恢复模式在发生故障下的任务重启时间，验证了 Flink-BQBS 能够在算子出现故障后快速恢复，从实验结果中得出，Flink-BQBS 在任务出现故障后重启的时间缩短为 Flink Restart-all 恢复模式的 1/6.

未来将在以下几个方面继续进行优化：1）将算法与 Flink 的 Restart-individual 和 Restart-all 这 2 种模式结合，进一步提升 Flink 在批流融合下的容错性能；2）进一步优化算法以降低额外资源消耗.

参考文献 (References):

[1] DEAN J, GHEMAWAT S. MapReduce: a flexible data processing tool [J]. **Communications of the ACM**, 2010, 53(1): 72–77.

[2] ZAHARIA M, CHOWDHURY M, FRANKLIN M J, et al. Spark: cluster computing with working sets [J]. **HotCloud**, 2010, 10: 95.

[3] KATSIFODIMOS A, SCHELTER S. Apache Flink: stream analytics at scale [C]// **IEEE International Conference on Cloud Engineering Workshop.** Berlin: IC2EW, 2016: 28–38.

[4] CHINTAPALLI S, DAGIT D, EVANS B, et al. Benchmarking streaming computation engines: storm, flink and spark streaming [C]// **2016 IEEE International Parallel and Distributed Processing Symposium Worshops.** Chicago: IPDPSW, 2016: 1789–1792.

[5] CHANDY K M, LAMPORT L. Distributed snapshots: determining global states of a distributed system [J]. **ACM Transactions on Computer Systems**, 2016, 3(1): 63–75.

[6] BALAZINSKA M, BALAKRISHNAN H, MADDEN S R, et al. Fault-tolerance in the Borealis distributed stream processing system [J]. **ACM Transactions on Database Systems**, 2008, 33(1): 81–124.

[7] HWANG J H, CETINTEMEL U, ZDONIK S. Fast and highly-available stream processing over wide area networks [C]// **2008 IEEE 24th International Conference on Data Engineering.** Cancun: ICDE**,** 2008: 804–813.

[8] HEINZE T, ZIA M, KRAHN R, et al. An adaptive replication scheme for elastic data stream processing systems [C]// **Proceedings of the 9th ACM International Conference on Distributed Event-based Systems**. Oslo: DEBS, 2015 : 150–161.

[9] CHANDRAMOULI B, GOLDSTEIN J. Shrink: prescribing resiliency solutions for streaming [J]. **Proceedings of the VLDB Endowment**, 2017, 10(5): 505–516.

[10] BENOIT A, RAINA S K, ROBERT Y. Efficient checkpoint/verification patterns [J]. **International Journal of High Performance Computing Applications**, 2017, 31(1): 52–65.

[11] AKBER S M A , CHEN H , WANG Y , et al. Minimizing overheads of checkpoints in distributed stream processing systems [C]// **2018 IEEE 7th International Conference on Cloud Networking.** Tokyo: CloudNet, 2018: 1–4.

[12] ZHUANG Y, WEI X, LI H, et al. An optimal checkpointing model with online OCI adjustment for stream processing applications [C]// **International Conference on Computer Communication and Networks.** Hangzhou**:** ICCCN, 2018: 1–9.

[13] LOMBARDI F, ANIELLO L, BONOMI S, et al. Elastic symbiotic scaling of operators and resources in stream processing systems [J]. **IEEE Transactions on Parallel and Distributed Systems**, 2018, 29(99): 572–585.

[14] 刘智亮. 面向流数据处理的动态自适应检查点机制研究 [D]. 吉林: 吉林大学, 2017.
LIU Zhi-liang . Research on adaptive checkpoint mechanism for large-scale streaming data processing [D]. Jilin: Jilin University, 2017.

[15] 郭文鹏, 赵宇海, 王国仁, 等. 面向 Flink 迭代计算的高效容错处理技术 [J]. 计算机学报, 2020, 43(11): 2101–2118.
GUO Wen-peng, ZHAO Yu-hai, WANG Guo-ren, et al. Efficient fault-tolerant processing technology for Flink iterative computing [J]. **Chinese Journal of Computers**, 2020, 43(11): 2101–2118.

[16] HAN D Z, CHEN X G, LEI Y X, et al. Real-time data analysis system based on Spark Streaming and its application [J]. **Journal of Computer Applications**, 2017, 37(5): 1263–1269.

[17] CARBONE P, EWEN S, FORA G, et al. State management in Apache Flink: consistent stateful distributed stream processing [J]. **Proceedings of the VLDB Endowment**, 2017, 10(12): 1718–1729.