Chapter 3. The Architecture of Apache Flink （flink 结构）

1. 一、System Architecture

   （系统结构）

1.flink 是一个有状态的分布式流处理引擎

可以与下列组件配合使用

*资源管理器：yarn、mesos、Kubernetes

*持久分布式存储：hdfs、s3

*高可用 HA　leader 选举：zookeeper

2. 二、Components of a Flink Setup

   （flink 的组件）

2.1. 前言：flink 组成的四个组件 JobManager，ResourceManager，TaskManager 和 Dispatcher

JobManager：

*每一个应用程序对应一个 JobManager

*JM 接收到一个应用程序，它包含一个叫做 JobGraph（logic data flow）以及做需要的所有依赖，进而将 JobGraph 转化为 ExecutionGraph（physical data flow），它包含了并行执行的 task

*JM 向 ResourceManager 申请资源（TaskManager slot），达到申请资源后，JM 将 ExecutionGraph 发送给 TaskManager 去执行

*执行期间 JM 负责所有的协调工作

ResourceManager：

*为 JM 申请 task maneger slot，如果不够会向资源管理者启动新的 slot

*JM 和 AppMaster 在同一进程

TaskManager：

*在 RM 的调度下为 JM 提供插槽运行 task

*与运行同一任务的 TaskManager 交换数据

Dispatcher：

*跨作业运行的概念，通过 restful 接口向 Dp 提交程序

*收到一个应用程序后，启动一个 JM，并把程序发送给 JM

*还提供 web 页面有关程序运行的信息

提交任务时的组件交互

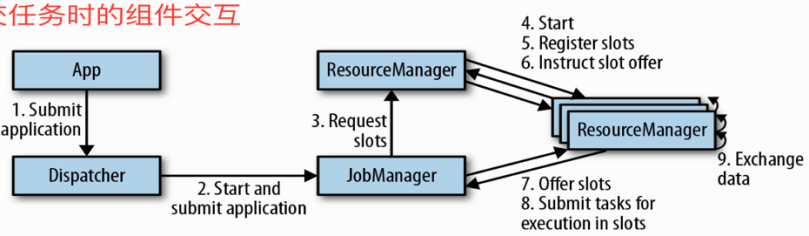


Figure 3-1. Application submission and component interactions

提交程序后的组件交互草图，不同模式略有不同

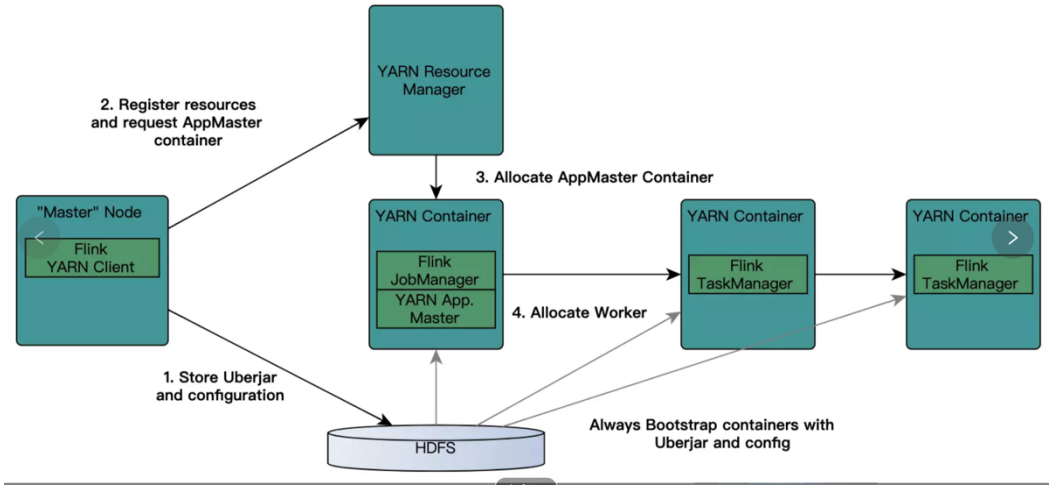
扩展补充：

flink on yarn 模式分两种（组件交互如图所示）

(1) flink on yarn session：先在 yarn 启动 flink 集群，再去执行 job，接着向 yarn 申请一块固定空间，资源被消耗完时，job 需要等待，使用测试环境

(2) flink on yarn　run：job 直接提交在 yarn 上运行，job 之间不会互相影响，资源独立

扩展补充　flink on yarn 提交模式



3. 三、Application Deployment

（应用程序部署）

可以有两种不同的方式运行

1.framework　2.lib

3.1. 1.framework 方式（submit 到服务）

将任务 jar 包通过 client submit 到一个服务，该服务可以是 flink Dispatcher、flink JobManager、yarn RM，如果直接提交给 JM 应用程序会马上运行，交给其他两个会先启动 JM 并移交程序给 JM 再执行

2.lib 方式是使用 doker 运行 flink，类似微服务的方式，第 10 章具体讲了在容器化环境的中的部署

（第 10 章的章节位置 Bundling and Deploying Applications in Containers）

4. **四、Task Execution**

（！！！！资源调度需深入研究、参考 flink meet up 的 ppt，1.9 在资源分配上也有变化）

4.1. 第一段：

TaskManager 可以同时执行多个任务。 这些任务可以是相同运算符（数据并行性），不同运算符（任务并行

性）甚至是不同应用程序（作业并行性）的子任务，TM 通过确定数量的 slot 来限制同时运行的 task 数量，一个 slot 仅执行 一个程序的一个算子的一个并行 task

第二段：

如图，根据 task 的最大并行度设置 slot 的数量，多个 task(线程)在相同的 TM(进程)可以避免网络传输更快的交换数据

扩展：spark 中的 task 也是线程，hadoop 的 map reduce task 是进程级别的

第三段：

TaskManager 在同一 JVM 进程中执行多线程任务。 线程比单独的进程更轻便，并且通信成本更低，但是不会严格地将任务彼此隔离。 因此，一个行为异常的任务可以杀死整个 TaskManager 进程以及在其上运行的所有任务。 通过为每个 TaskManager 仅配置一个插槽，您可以跨 TaskManager 隔离应用程序

总结：

1.TM 按照最大的 task 并行性设置 slot 的个数

2.TM 内的并行度可以减少数据在网络中的交换，但由于 task 都运行在相同的 TM 进程中，一个 task 出错就会导致 TM 挂掉，所以过多的并行度也会增加 TM 挂掉的可能性，

3.一个 TM 里面只有一个 slot 不会出现上述问题，所以并行性需要适当，并不是越多越好

4.不同 job 的 task 也可能运行在同一个 TM 中，作业并行性

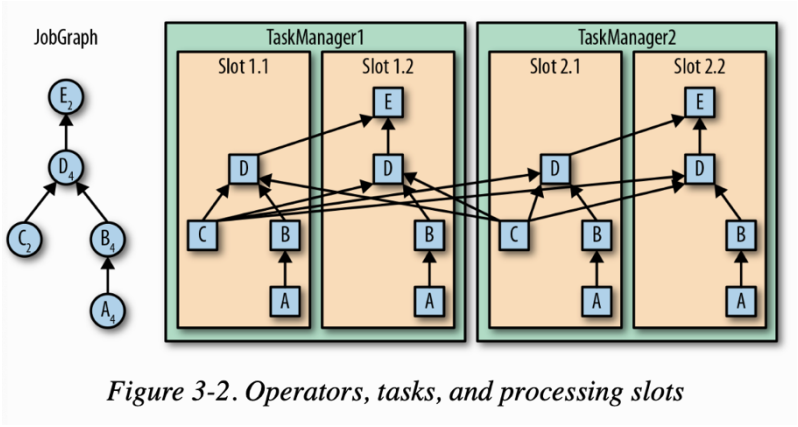


Figure 3-2. Operators, tasks, and processing slots

5. 五、Highly Available Setup

（flink 故障恢复）

5.1. 前言：

流程序一般都是 7*24 小时运行， 所以重要的是即使有进程失败也不要停止运行，从故障中恢复，首先需要重启失败的进程，再重启程序恢复状态，本节将介绍如何重启失败的进程

1.TaskManager failures

如果运行过程中某一个 TM 失败，则 JobManager 将要求 ResourceManager 提供更多的 slot，如果无法满足，JM 将无法启动程序，启动策略决定了 JM 重启的频率和间隔

2.JobManager failures

JM 高可用使用的是 zookeeper 来保证的，JM 将 task check point 句柄信息存在远程存储，并将存储位置的指针发送给 zookeeper 以此来达到切换 JM 的目的

切换 JM 的步骤 如图 3-3

*请求 ZK 获取 JobGraph、jar、及 task 最后一个状态的检查句柄

*从 ResourceManager 获取 slot 以便继续执行任务

*重新启动应用程序并将所有 task 重新设置为最后的 checkpoint

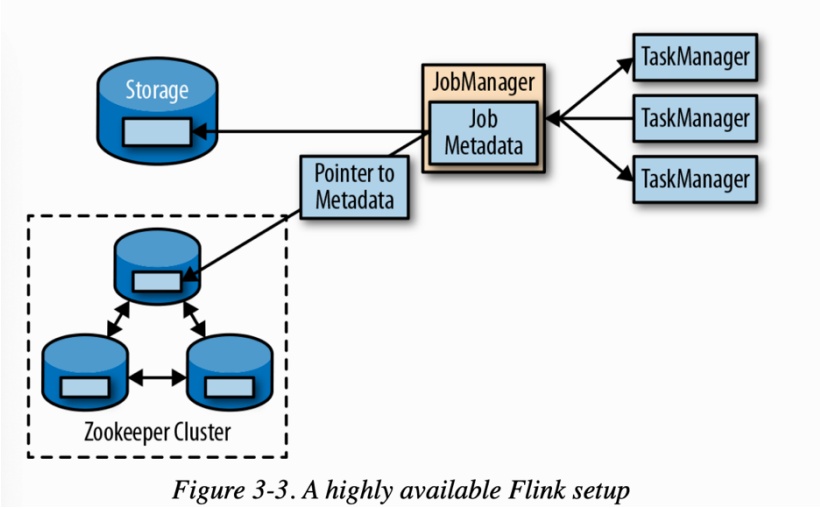


Figure 3-3. A highly available Flink setup

3.k8s、yarn、standalone 模式下的高可用

*k8s 通过容器编排重启 JM 或者 TM

*yarn 通过剩余的进程触发 JM 或者 TM 重启

*standalone 模式目前不提供失败进程重启，所以执行一个 standby 的 JM 或者 TM 会很有用

6. Data Transfer in Flink

（数据在 flink 中的传输）

6.1. 前言：

程序中的 task 是持续交换数据的，TM 负责将数据从 发出的 task 发送到 接收的 task，在传送 record 之前 TM 将 reecord 缓存在它的网络组件中，分批传送，该机制有效利用了网络资源并作为高吞吐的基础，类似于网络或磁盘 I/O 的缓存技术

注：这也侧面说明了 flink 的模型确实是微批的，env.setBufferTimeout(timeoutMillis)可以设置缓冲区的超时时间，默认 100ms，setBufferTimeout(-1)表示缓冲区满了再 flush

1.task 的数据交换

*network buffer pool：

每一个 TM 都有一个 network buffer pool 用来发送和接收数据，默认大小为 32k，taskmanager.memory.segment-size 参数可以调整大小，

*不同 TM 之间通信：

每一对 TM 之间都会有一个永久的 TCP 连接，TCP 连接采用的是多路复用机制，如果 sender task 和 reciver task 在不同的 TM，则他们之间通过操作系统的网络堆栈进行通信

*发生 shuffle 或者广播时，需要为每一个接收的 task 建立缓冲区，缓冲区的数据是所涉及到的 task 的平方数

*在同一个 TM 中的数据交换不会涉及到网络传输

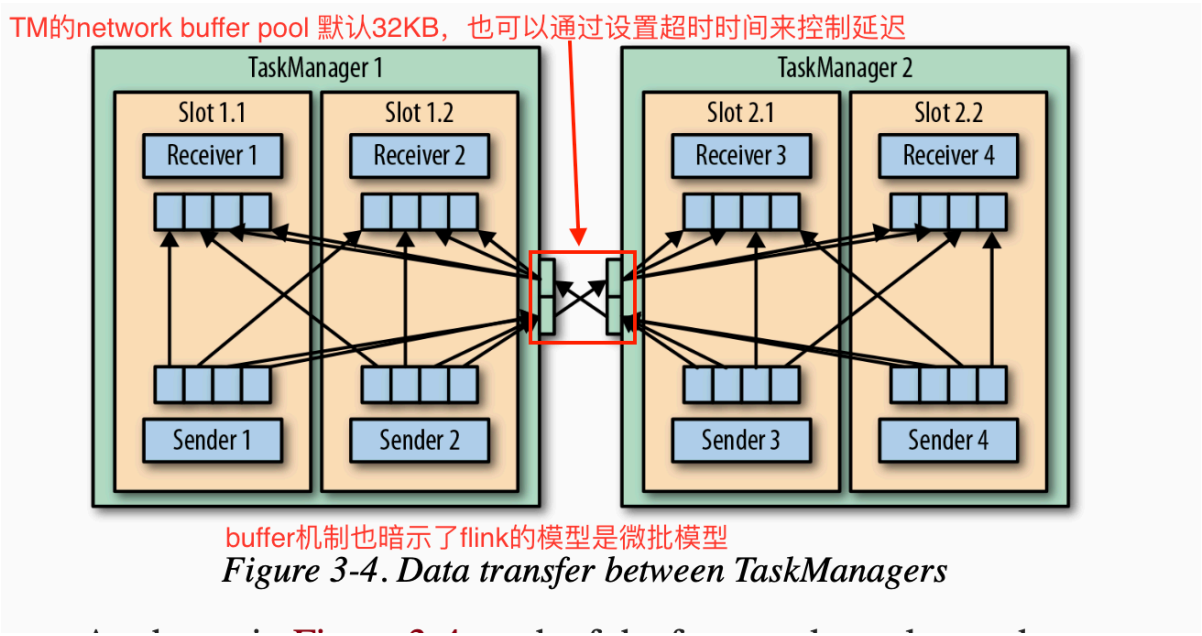


Figure 3-4. Data transfer between TaskManagers

参考链接：

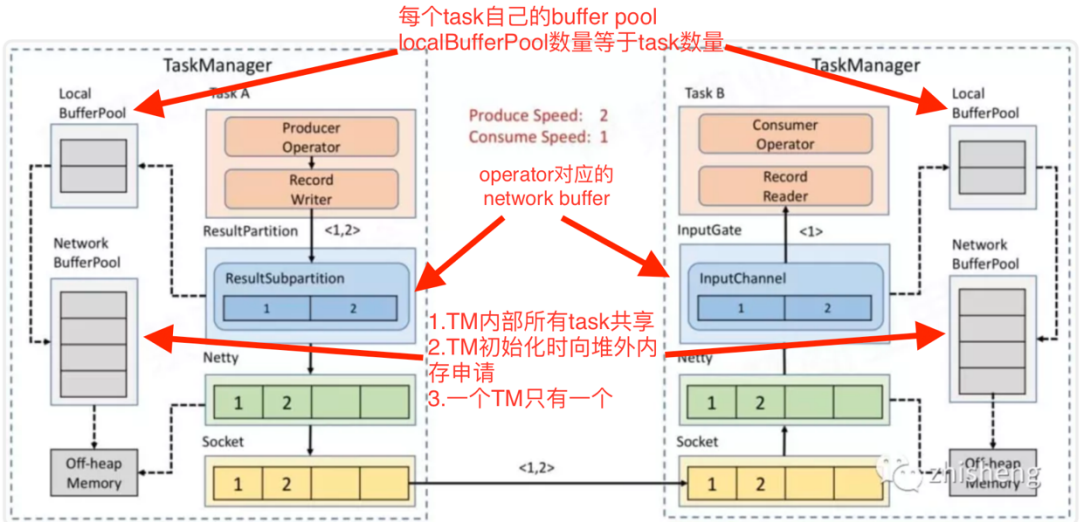
（深入研究请查关键词　flink 网络堆栈或者 task 数据交换原理）

*https://flink.apache.org/2019/06/05/flink-network-stack.html

*https://cwiki.apache.org/confluence/display/FLINK/Data+exchange+between+tasks

*数据传输和反压详情可以参考收藏的 pdf（一文彻底搞懂 Flink 网络流控与反压机制）

*TM 的各种 buffer 如图所示



2.Credit-Based Flow Control （基于信用的流量控制）

a 背压：

当下游的某一个 task 消费速度跟不上接收速度时，接收端的数据会被堆满，TCP 的读取会暂停，继而影响发送端的数据池也会堆满数据，1.5 之前的版本没有直接处理反压，因为数据传输环节中，反压会向上游传

递直到 source 降低读取速率

影响：1.5 之前的机制会导致不相关的 task 也受到反压的影响，因为 tcp 链接是被多路复用的（多 task 共同使用）

b 基于信用的机制：

为了解决精细化的解决背压问题，上游发送数据时会发送上游的数据和数据量 backlog size，下游会向上游反馈现在的空闲 buffer 有多少(credit 分数)，此机制作用在应用层，也就是 operator 对应的 network buffer（ResultSubPartition 和 InputChannal），这样可以保证公用的 tcp 和 netty 层不受阻塞，其他的 subtask 正常运行，当信用分数 0 时，上游 task 不会向 netty 中发送数据，也就不会造成 tcp 阻塞

解决的问题：

*精细化背压问题，不会由于某一个单独的 task 导致整个 TCP 连接阻塞

*有效解决数据倾斜的问题，存在背压的 task 不会再发送数据

*将 buffer 直接添加到队列里面，发送数据将更贴近网络状况，繁忙时继续往 buffer 中写数据，不繁忙时可以随时把 buffer 中的数据发送出去

*对于 ckp barrier 不存在反压时会立即 flush，存在反压时也需要等待 flush

3.Task Chaining

task chain 的条件

*相关算子并行度相同

*相关算子通过前向策略交换数据

原理：将相关运算符合并在同一个线程中执行，减少了序列化与反序列化的开销，但昂贵的操作不建议使用 task chain ， 因为这样会在同一个 slot 中做过多的计算

默认：开启

task chain 的条件

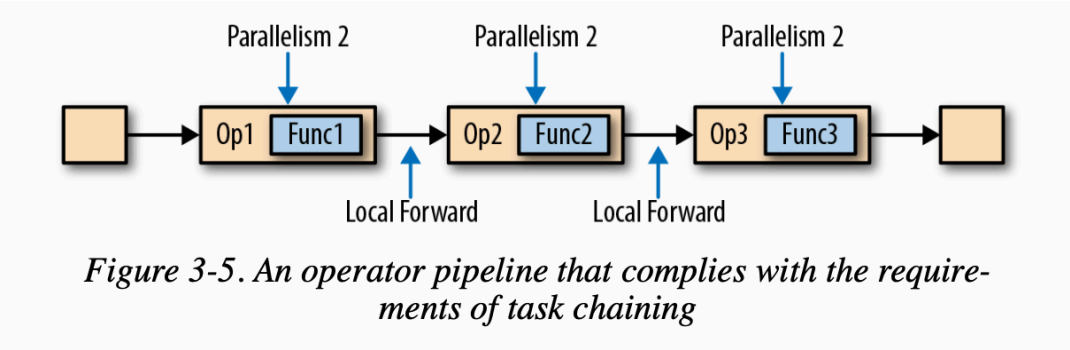


Figure 3-5. An operator pipeline that complies with the requirements of task chaining

task chain 后的执行视图

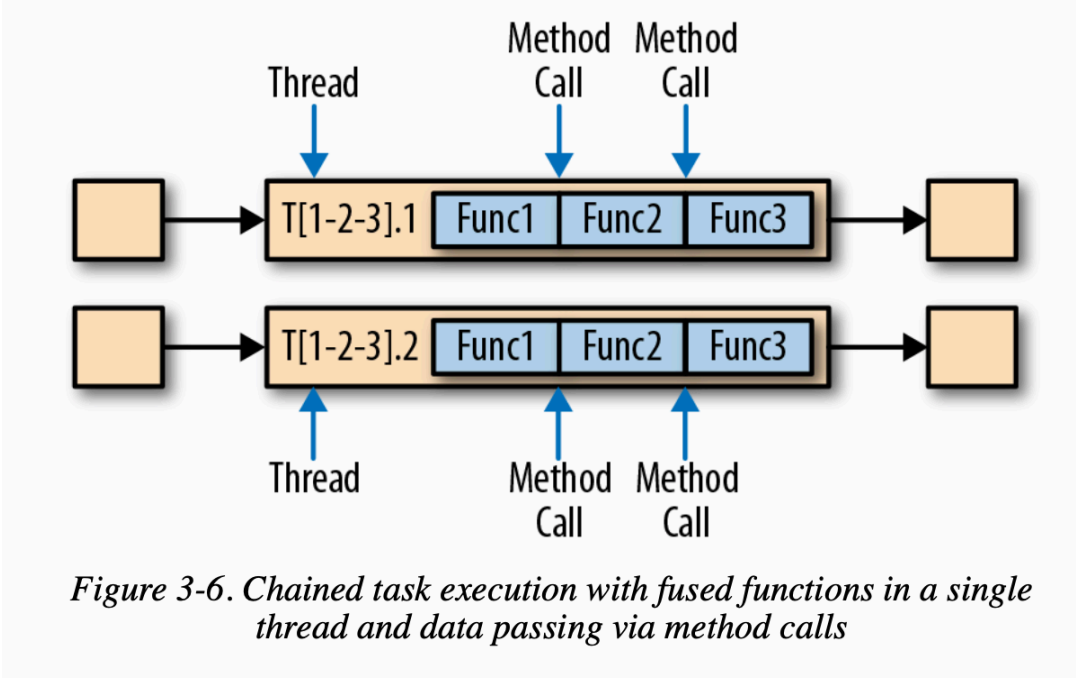


Figure 3-6. Chained task execution with fused functions in a single thread and data passing via method calls

7. Event-Time Processing

7.1. 前言：基于事件时间处理数据比较复杂，将介绍如何使用时间戳和水印来处理事件时间

1.Timestamps

Flink 处理 event-time 语义，每条记录都需要伴有时间戳，时间戳关联一条指定时间点的记录，时间戳编码为 16 字节的 Long 值，并将它们作为元数据附加到记录中，基于事件时间的算子使用的就是时间戳来触发计算

2.Watermarks

用于提供给每个 task 当前的事件时间（个人理解就是基线），用来做窗口计算的触发条件

*必须是单调增加的

*WM 的事件戳是 T，则代表后续记录的时间戳都是大于 T 的，后续记录小于 T 的事件称为后期记录，将在 ‘Handling Late Data’ 一章介绍不同方法

*紧密的水印延迟低，但可能结果不完整，保守的水印延迟较大

举例：窗口划分 0 ~ 10s、10 ~ 20　假如 wm 设定值=5s，每 10s 触发一次计算

事件　8s、9s、12s、10s、13s

WM = max（8s）- 设定值（5s）= 3s < 10s 所以不会触发 windows 计算

WM = max（8s，9s）- 设定值（5s）= 4s < 10s 所以不会触发 windows 计算

WM = max（8s，9s，12s）- 设定值（5s）= 7s < 10s 所以不会触发 windows 计算

延迟：WM = max（8s，9s，12s，10s）- 设定值（5s）= 7s < 10s 所以不会触发 windows 计算　延迟事件的到来并没有影响 window 的计算 这也是 WM 的意义

3.Watermark Propagation and Event Time

（水印传播和事件时间）

*WM 被实现为一个特殊的 record，它可以被 task 发送或者接收

*task 收到水印后

task 内部有一个 time service，用来注册 timer

(1)更新自己的内部时钟——>(2)task 的 time service 会识别所有小于当前水印的 timer，并触发相应的计算函数——>(3)task 会发出更新后的水印

*wm 的传播和 task 内部时间更新如图 3-9

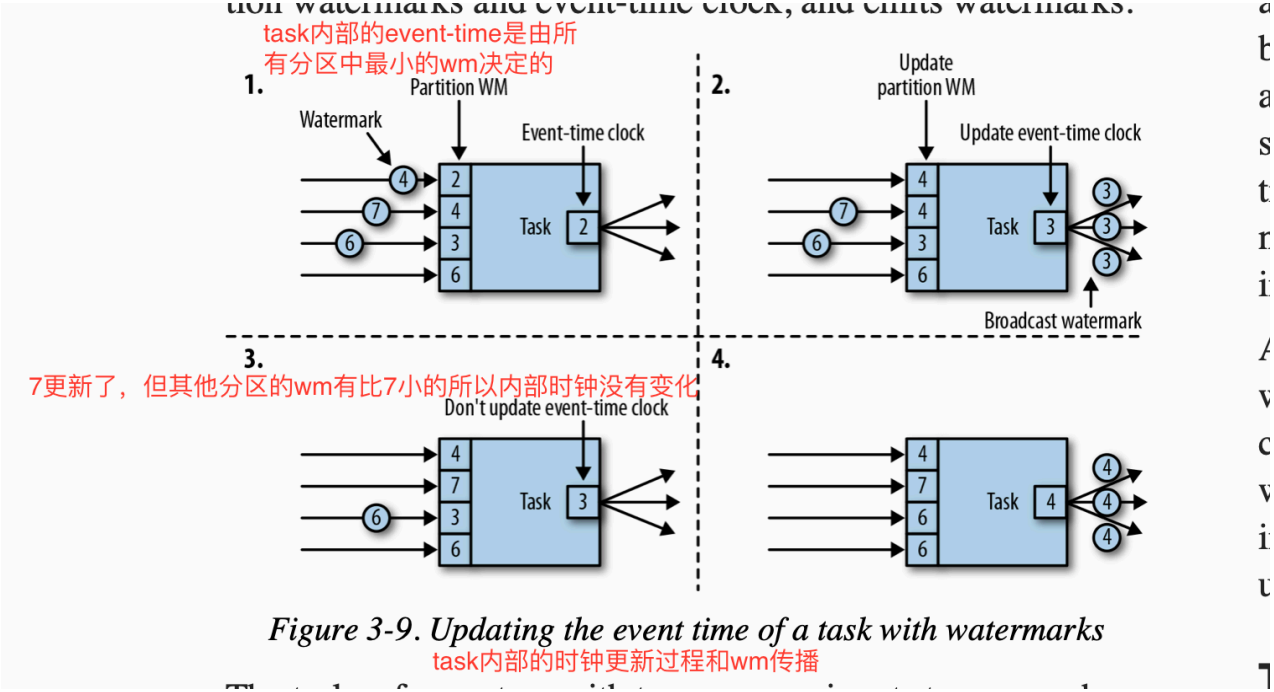


Figure 3-9. Updating the event time of a task with watermarks

task内部的时钟更新过程和wm传播

*task 发出 record 和 wm 的前提

所有分区都不断提供不断增加的水印。一旦一个分区不前进其水印或变得完全空闲并且不发送任何记录或水印，计算也不会被触发，相似情况发生在两个明显不同的 wm 的流，一般都是将快流存入状态

4.Timestamp Assignment and Watermark Generation （时间戳分配和水印生成）

*三种方式分配时间戳和 WM

第一种：通过源函数，如果源函数不再发出水印，则状态定义为 idle，下游计算水印会排出空闲分区

第二种：定期分配器：调用 AssignerWithPeriodicWatermarks 函数

第三种：标点分配器：AssignerWithPunctuatedWatermarks，与 AssignerWithPeriodicWatermarks 函数相反，此函数可以但不需要从每个记录中提取水印

后言：用户自定义的分配函数尽量在靠近源的时候去使用，因为在操作符处理完记录之后，很难推断出记录的顺序及其时间戳记。这也是为什么在流应用程序中间覆盖现有的时间戳和水印不是一个好主意的原因，尽管用户定义的函数可以做到这一点

8. 六、State Management

讨论 Flink 支持的不同类型的状态。 将说明状态后端如何存储和维护状态，有两种状态类型 operator state and keyed state

8.1. 1.Operator State

(会绑定到一个特定的 operator,其状态是属于一个 operator 的)

由同一并行 task 处理的所有记录都可以访问 operator state。 相同或不同 operator 的另一任务无法访问 operator state

Flink 提供了三种用于 operator 状态的原语

*List state ：将状态表示为条目列表

*Union list state：也是条目列表，区别在于如何在故障时还原

*Broadcast state：专为 operator 每个 task 的状态相同的特殊情况而设计。可以在检查点期间和重新缩放运算符时利用此属性

2.Keyed State(相当于分组后的 operator state)

前言：如图

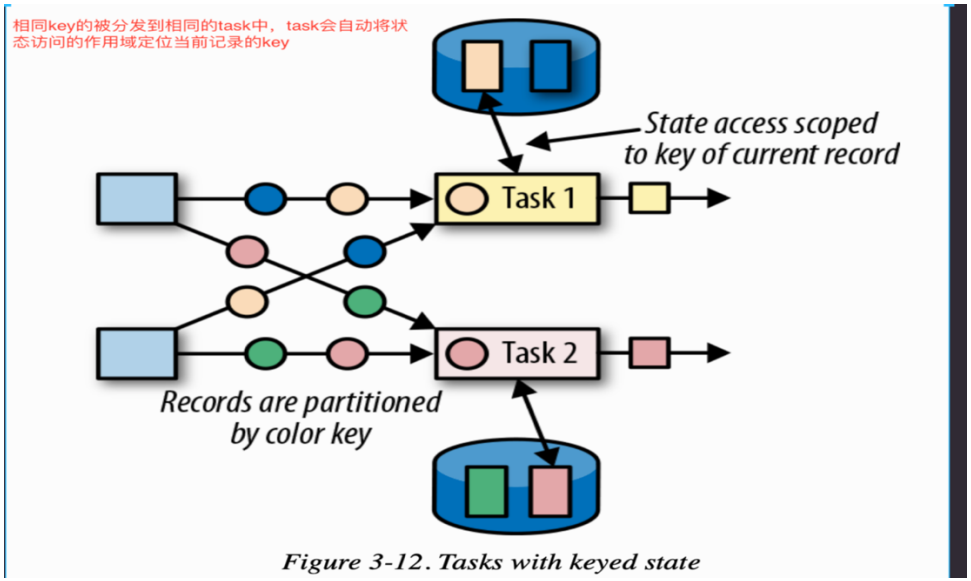


Figure 3-12. Tasks with keyed state

三种状态存储的类型

*value 状态：将 key 的状态存储为单一的值

*list 状态：将每一个 key 的值存为 list，list 可以是任何类型

*key-value 形式：存储键值对

3.State Backends（一共有三种，不同方式决定了 state 的存储和）

参考：https://blog.csdn.net/jmx_bigdata/article/details/100037158

*每一个并行 task 都将在本地维护其状态以确保快速的状态访问。状态的精确存储，访问和维护方式由可插拔组件（称为 State Backend）确定。

*一个 State Backend 负责两件事：

本地状态管理

将状态指向远程位置的 check point

*flink 的提供的 state backend 一个将状态键值对存储在 jvm Heap 中，另一个将比较大的状态写进 RackDB 中，然后将其写入本地磁盘，后者访问较慢，但存储的状态较大

*state backend 负责将任务 checkpoint 指向远程持久存储。

远程存储可以是分布式文件系统或数据库系统，rackDb 支持增量更新

4.Scaling Stateful Operators(扩展有状态的算子)

不同类型的状态有不同的扩展方式,Flink 支持四种用于扩展不同状态类型的模式

*keyed state ---> 将 key 重新分区，使用 key group 来保证以前的数据尽量分在和以前相同的分区

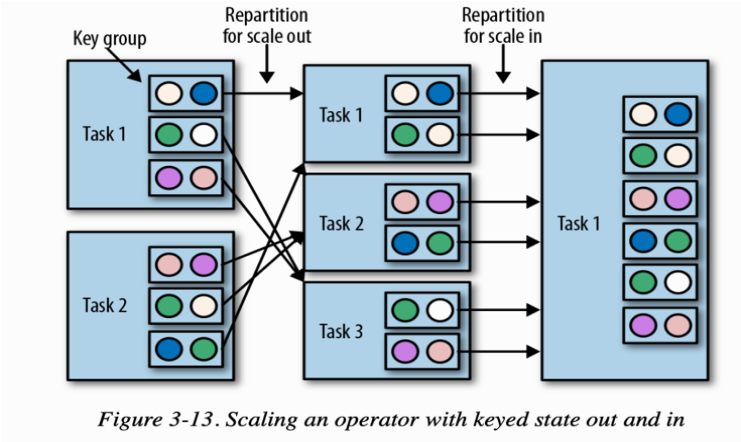


Figure 3-13. Scaling an operator with keyed state out and in

*operator list state--->将之前的 list state 均匀的非配在扩展后的 task 上

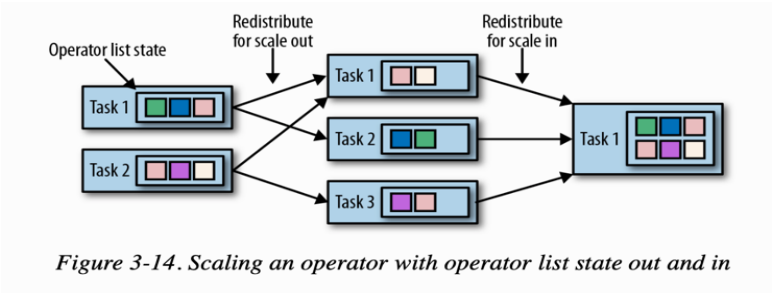


Figure 3-14. Scaling an operator with operator list state out and in

*operator union list state---->向每个任务广播状态条目的完整列表，任务可以选择要使用的条目和要丢弃的条目

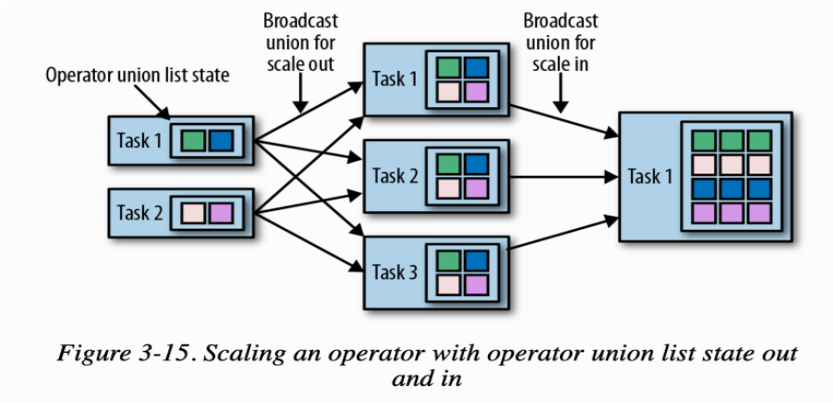


Figure 3-15. Scaling an operator with operator union list state out and in

*operator broadcast state--->将状态复制到新任务来扩大规模,在缩小规模的情况下，多余的任务将被简单地取消

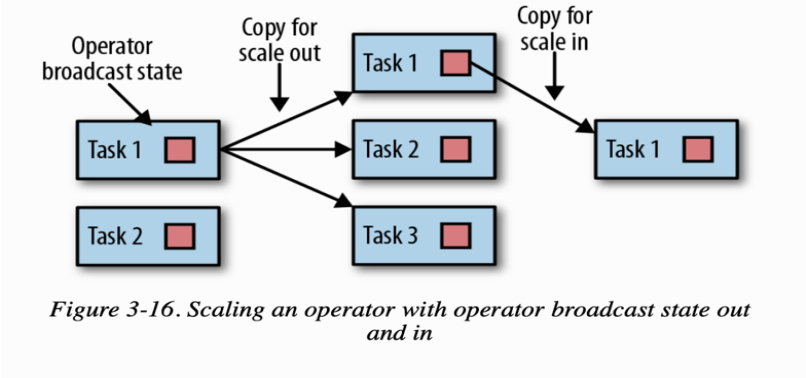


Figure 3-16. Scaling an operator with operator broadcast state out and in

9. 七、Checkpoints, Savepoints, and State Recovery

介绍 Flink 的检查点和恢复机制以及独特的保存点功能

9.1. 前言：flink 作为分布式计算框架，必须考虑失败后处理机制，本节将介绍 flink 的恢复机制和 exactly-once 状态保障

1.一致的检查点

a 恢复机制的前提：

基于状态一致的 checkpoint，对于有状态的流应用程序，它的一致 checkpoint 的含义是所有准确处理过输入的所有 task 的状态副本

b 一般的 naive algorithm（朴素）算法过程如下

暂停所有输入流的读取—>等待所有 in-flight 数据处理完毕—>所有的 task 将状态远程 copy 持久化存储中，则 check point 完成—>恢复所有的流的读取

flink 并没有采取上述算法

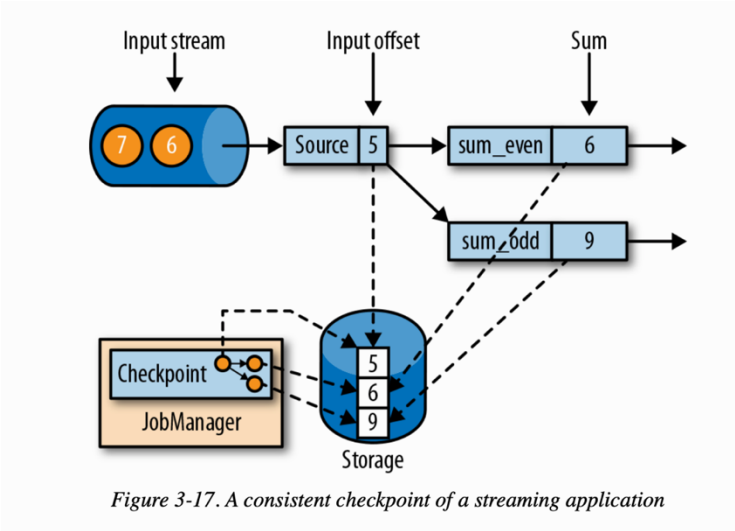


Figure 3-17. A consistent checkpoint of a streaming application

2.从一致的检查点恢复 如图 3-18

*flink 会定期获取程序的 check point，发生故障会使用最新的 check point 来恢复任务

*故障恢复的步骤 ：重启整个应用程序——>重置所有的 task 的状态为最新 check point 的状态——>恢复所有任务的处理

*状态一致性只能在流量重放的场景下实现

*flink 的状态一致性指的是程序和系统内部的状态，并不保证已经写入下游的结果数据，flink 提供了仅写一次

的 sink，例如在 check point 完成时再提交结果数据，对于存储系统来说可以使用幂等更新来实现

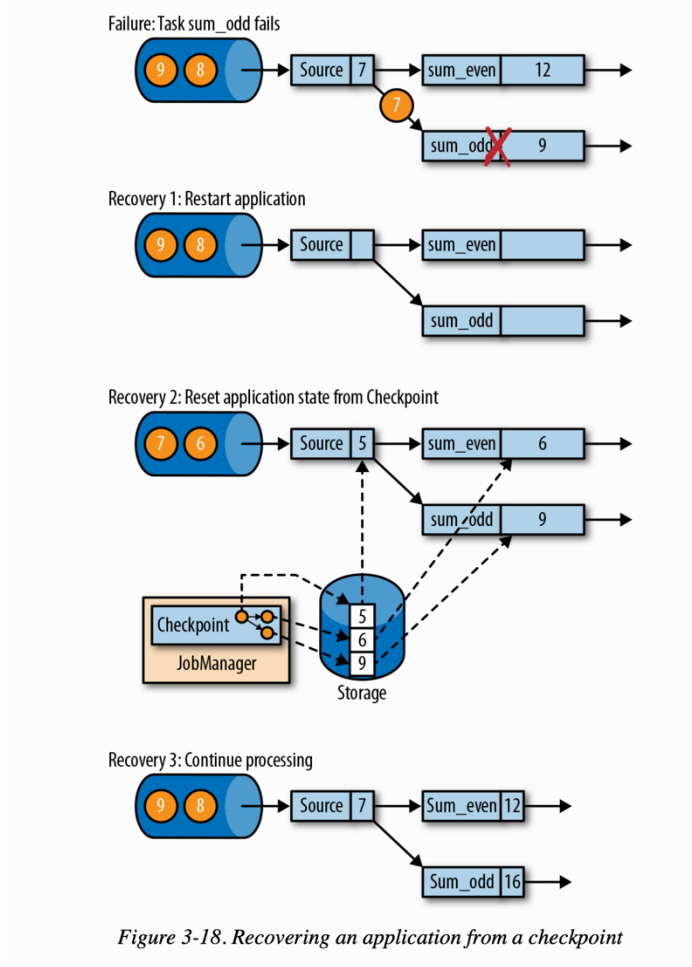


Figure 3-18. Recovering an application from a checkpoint

3.flink 使用的 check point 算法

前言：传统的 navie algorithm 会出现 stop world 的现象，对于延迟较低的应用不能容忍，所以 flink 使用的是

Chandy–Lamport algorithm

check point barrier 是 source oprator 注入到数据流中的特殊记录，check point barrier 带着 check point ID 来区分属于哪个 check point 并把数据流逻辑分为两部分，barrier 之前的所有状态修改均包含在 barrier 的 check point 中，barrier 之后的 records 导致的所有修改均包含在后续 check point 中

举例说明 check point 的原理

（1）JobManager 通过向 source task 发送 barrier 初始化一个 check point

（2）task 收到 barrier 之后在 state backend 触发 local state checkpoint——>完成 checkpoint 之后 state backend 会通知 task，task 进而向 JM 确认 checkpoint 完成(通过注入 barrier，source 函数可以定义发生 checkpoint 的位置)

（3）在 barrier 向下游传递过程中，下游 task 需要收到所有分区的 barrier，当收到其中一个时，则当前分区的 records 会缓存起来，等待其他分区的 barrier，没有收到 barrier 的分区的数据会继续处理，等待 barrier 的过程叫做 barrier 对齐，对齐的目的是为了保证 Exactly-Once 模式，At-Least-Once 不需要对齐，Exactly-Once 模式目的是在故障恢复时不重复消费数据

4.Performace Implications of Checkpointing （checkpoint 的性能含义）

*第一段：flink checkpointing 算法会增加延迟，但 flink 做了一些调整来缓解这种延迟

*第二段：（减少延迟的第一种方法是异步线程 copy 本地状态来减少延迟的过程）

flink 的 checkpoint 是 state backend 来实施的，FileSystem state backend 和 RocksDB state backend 支持异步 checkpoint，当 checkpoint 被触发时——>上述两种 statebackend 将会先在本地 copy 一份状态——>之后异步线程再 copy 状态到远程存储并在完成 checkpoint 通知 task。（RocksDB state backend 支持增量 checkpoint）

*第三段：（减少延迟的第二种方法是配置 barrier 不对齐方式即 at-least-once）

5.Savepoints（保存点）

*前言：和 checkpoint 机制类似，但需要手动配置，主要作用是版本更新，bug 修复做到无感知切换

*Using savepoints（使用保存点）

checkpoint 主要用在特殊场景，相同集群相同配置，savepoint 可以用在如下场景，前提是应用程序和 saveoint 兼容

（1）bug 修复，A/B test

（2）使用不同的并行度启动相同的应用程序，以便根据实际情况用来扩展

（3）不同集群启动相同的代码，可以用来切换 flink'版本

（4）暂停应用程序用来释放集群资源

（5）使用 savepoint 存档来版本化应用程序

（6）实际使用中很多人根据 savepoint 来切换到廉价的数据中心

6.Starting an application from a savepoint（从 savepoint 启动一个应用程序）

*重新启动时根据 oprator ID 和 state name 匹配 oprator 的状态，savepoint 不包含 task 信息，因为并行度有可能改变

*如果程序代码发生改变，则对应的 oprator ID 有可能改变，因此状态匹配有可能出错，所以 flink 建议手动分配 oprator 的标识符

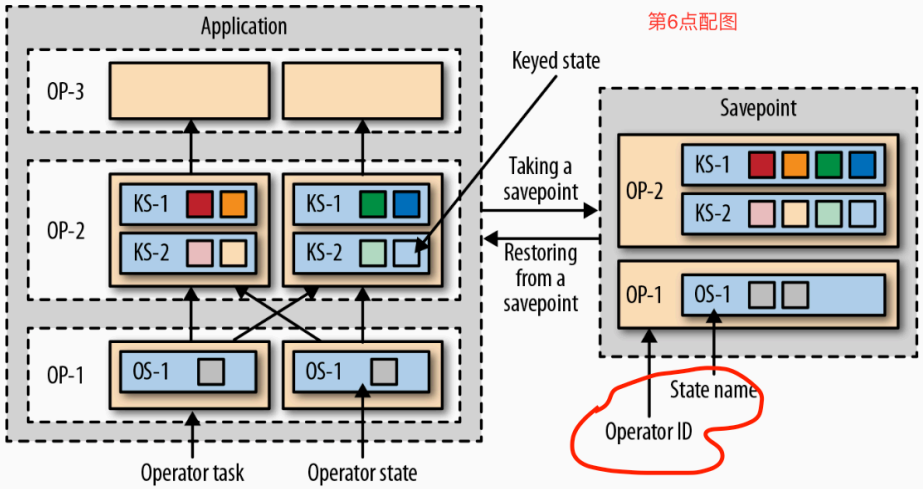


Figure 3-26. Taking a savepoint from an application and restoring an application from a savepoint

10. Summary

（1）高级体系结构

（2）网络堆栈

（3）事件时间处理模式

（4）状态管理和故障恢复