1 HDFS

1.1 体系架构

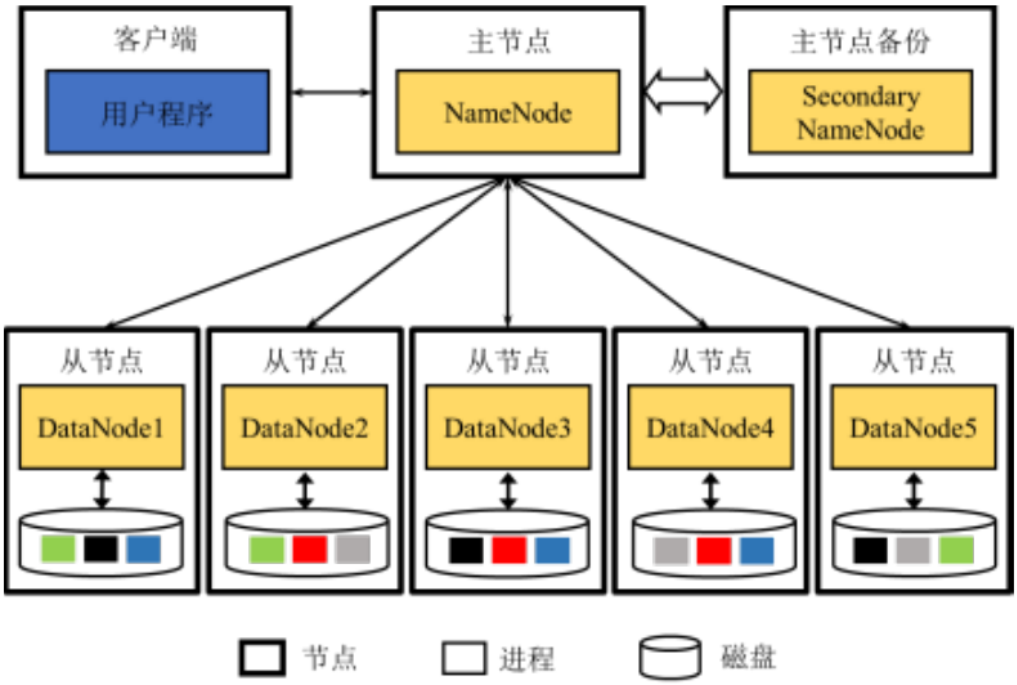

图 1.1: HDFS 架构

- NameNode: 元数据管理 (文件目录结构、位置), 维护 DataNode
  - SecondaryNameNode: 一旦 NameNode 发生故障时利用 SecondaryNameNode 进行恢复
- DataNode: 负责数据块的存储, 为客户端提供实际文件数据
  - Client 直连 DataNode

1.2 工作原理

- 当客户端读取数据时，从 NameNode 获得数据块不不同副本的存放位置列列表
- Write One Read Many: 一次写入多次读取 (只读文件系统, 避免读写冲突, 用户编程无需考虑文件锁)

1.2.1 备份机制

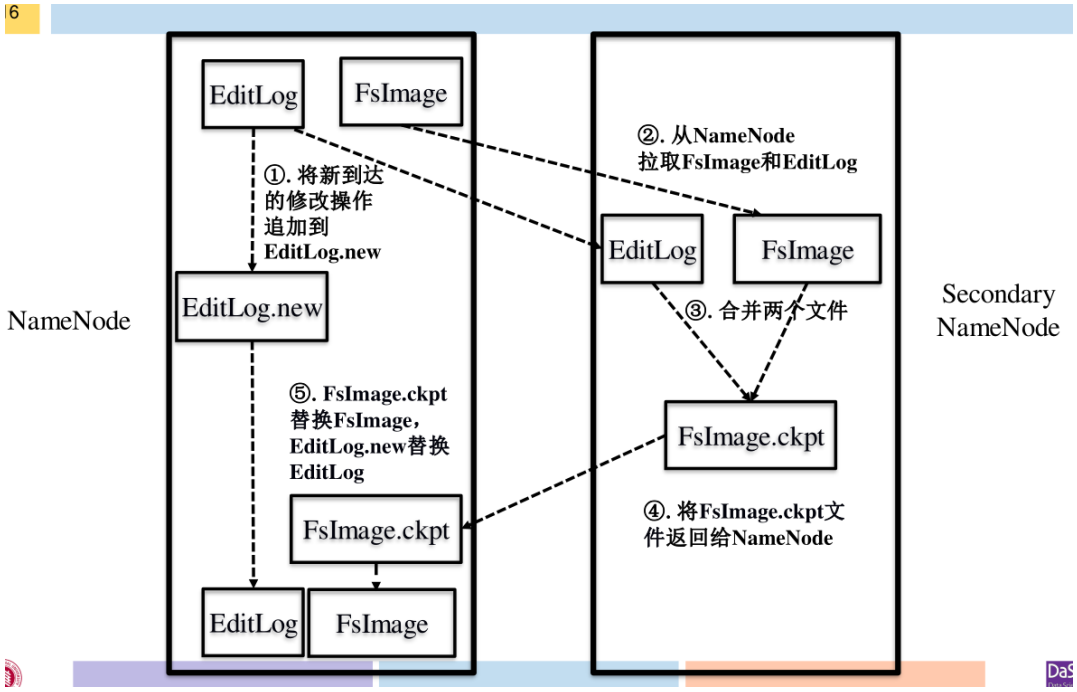

图 1.2: 备份

- FsImage：内存在文文件目目录结构及其元信息在磁盘上的快照
- EditLog：两次快照之间，针对目目录及文文件修改的操作

1.2.2 文件操作

- NameNode 成功后才成功
- DataNode 数据标记后延迟**删除**

1.2.3 副本存放策略

1. 就近 > 不太满不太忙 (快速写入, 同步)
2. 不同机架 (减少跨机架网络, 异步?)
3. 同机架不同节点 (应对交换机故障?, 异步?)

1.3 容错机制

1.3.1 NameNode 故障

根据 SecondaryNameNode 上的 FsImage 和 Editlog 进行恢复

1.3.2 DataNode 故障

节点上面面的所有数据都会被标记为“不可读”，数据块复制到其余节点

2 MapReduce

2.1 体系架构

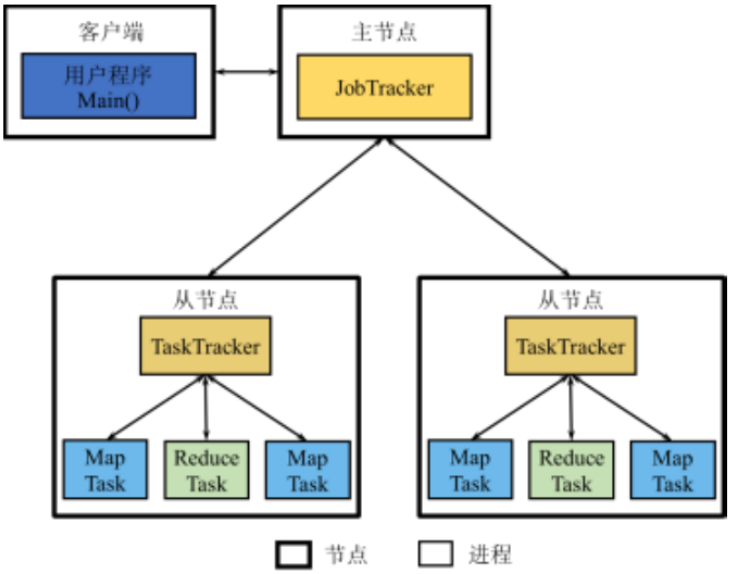

图 2.1: MapReduce 架构图

- JobTracker
  - 资源管理: 通过监控 TaskTracker 来管理系统拥有的计算资源
  - 作业管理: 负责将作业（Job）拆分成任务（Task），并进行任务调度以及跟踪任务的运行进度、资源使用量等信息
- TaskTracker
  - 管理本节点的资源: TaskTracker 使用 slot 等量划分本节点上的资源量（CPU、内存等）
  - 执行 JobTracker 的命令: 接收 JobTracker 发送过来的命令并执行（如启动新 Task、杀死 Task 等）
  - 向 JobTracker 心跳汇报情况: 通过心跳将本节点上资源使用情况和任务运行进度汇报给 JobTracker
- Child -> Task: 任务执行
- Client: 提交作业, 查看运行状态

2.1.1 MapReduce 2.0 (Yarn)

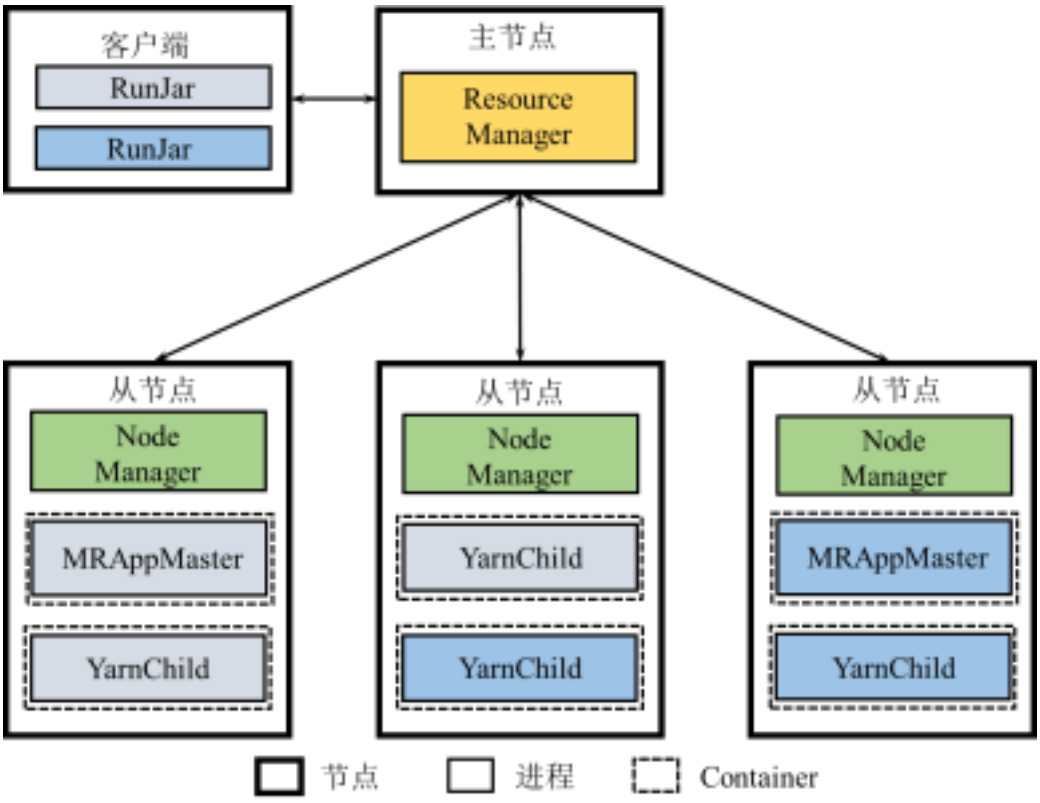

图 2.2: MapReduce 架构 (Yarn)

2.2 工作原理

- 计算节点向数据靠拢

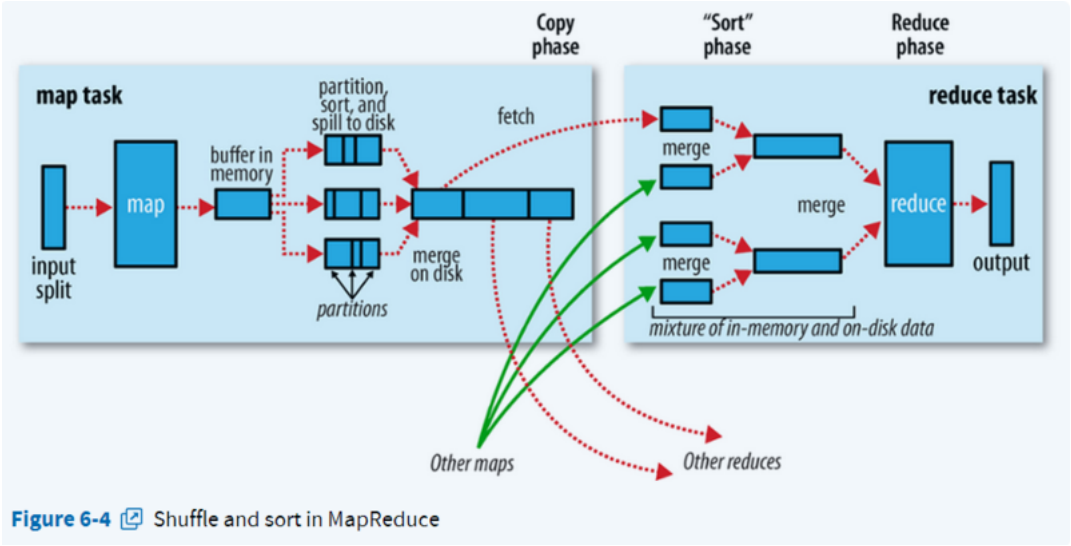

图 2.3: MapReduce 流程

2.2.1 输入格式

split: 根据文件格式的切片 (text, kv, line…), 可跨 block. 一个 split 一般对应一个 map, 即一般决定 map 数量

2.2.2 Map

```
map: [(k', v')] -> [(k, v)]
```

2.2.3 Shuffle

Shuffle 过程是指 `map` 的结果传入 `reduce` 的过程

```
shuffle: Mapper -> [(k, v)] -> ordered([(k, v)]) (-> combiner) -> Reducer -> merge
combiner: ordered([(k, v)]) -> ordered([(k, combine(v))])
```

1. map
    1. `[(k', v')]` 存入缓冲区
    2. 排序并写入磁盘
        1. spill: 达到缓冲区阈值, 锁定缓冲区, `sort(k')` 后写入本地磁盘
        2. spil 数达到阈值, 使用 combine 压缩
        3. merge: 合并有序 spill 后写入本地磁盘
2. reduce, 根据 k=key 的 offset 拉取 (copy) 数据同时 merge, 最后给 reduce 函数

2.2.4 Reduce

```
reduce: (k', [v']) -> (k', v')
```

2.3 容错机制

- JobTracker 故障: 未处理, 重新执行
- TaskTracker 故障: JobTracker 安排其他 TaskTracker
- Task 故障
    - Map 故障: 重新执行 Map
    - Reduce 故障: 重新执行 Reduce (取哪里重新读入数据?) 直至 max-try

2.4 局限性

- 编程框架的表达能力有限，用用户编程复杂 (Spark)
- 单个作业 Shuffle 阶段的数据以阻塞方方式传输，磁盘 IO 开销大大、延迟高高 (Spark)

- 多个作业之间衔接涉及 IO 开销，应用用程序的延迟高 (Spark)
- 资源管理与作业高度耦合 (Yarn)
- 作业控制管理高度集中，JobTracker 存在单点故障风险，内存开销大 (Yarn)

3 Spark

Spark 解决了 MapReduce 的局限.

- 数据模型 **R**esilient **D**istributed **D**ataset
  - Immutable
  - Resilient (弹性): 容错
  - RDD 算子
    - 创建 RDD
    - Transformation (转换)
    - Action (操作): 转换结束
- 计算模型 DAG (Job)
  - Operator DAG: 节点是算子

    DAG in Apache Spark is a combination of Vertices as well as Edges. In DAG vertices represent the RDDs and the edges represent the Operation to be applied on RDD. Every edge in DAG is directed from earlier to later in a sequence. When we call an Action, the created DAG is submitted to DAG Scheduler which further splits the graph into the stages of the task.
  - RDD Lineage: 节点是 RDD

    As we know, that whenever a series of transformations are performed on an RDD, they are not evaluated immediately, but lazily(Lazy Evaluation). When a new RDD has been created from an existing RDD, that new RDD contains a pointer to the parent RDD. Similarly, all the dependencies between the RDDs will be logged in a graph, rather than the actual data. This graph is called the lineage graph.

3.1 体系架构

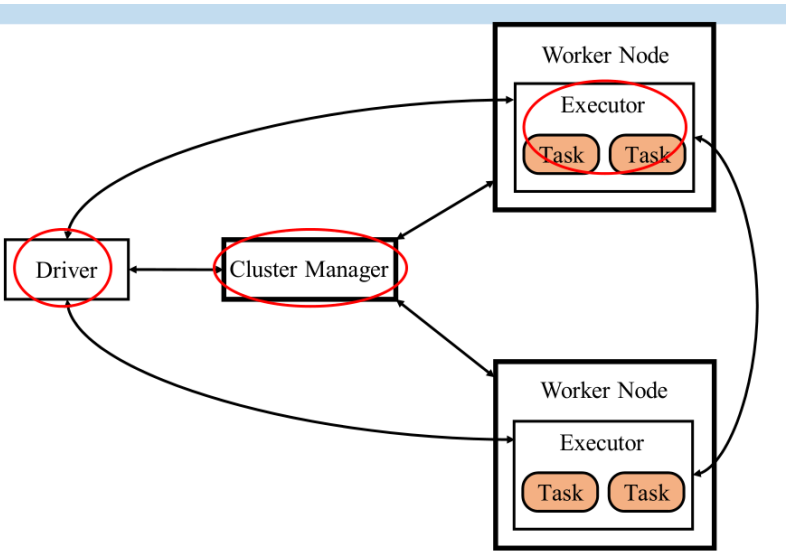

图 3.1: Spark 抽象架构图

Spark 的架构实现了资源管理 (Cluster Manager) 和作业管理 (Driver) 两大功能的分离

- Cluster Manager: 工作节点资源管理
- Driver: 负责启动应用程序的主方法并管理作业运行。逻辑上，Driver 独立于主节点、从节点以及客户端。

    1. 创建 SparkContext
    2. 向 CM 申请 WorkNode
    3. 通知 WorkNode 启动 Executor 并确认
    4. 创建 DAG(可能有多个) 并持久化 (用于恢复)
    5. 启动 Application, 交给 Executor 执行 (调度 Task, 监控 Job 进度)

- Executor: 负责任务执行, Executor 是运行在工作节点上的一个进程, 它启动若干个线程 Task 或线程组 TaskSet 来进行执行任务
- Task: 执行 Task 的**线程**（MapReduce 中的 Task 是进程）

3.1.1 Standalone

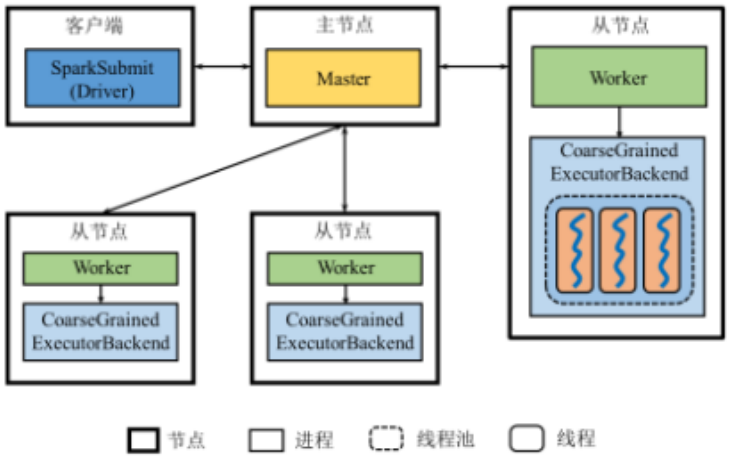

图 3.2: Spark Standalone Client Mode: Driver 集成在 Client 中

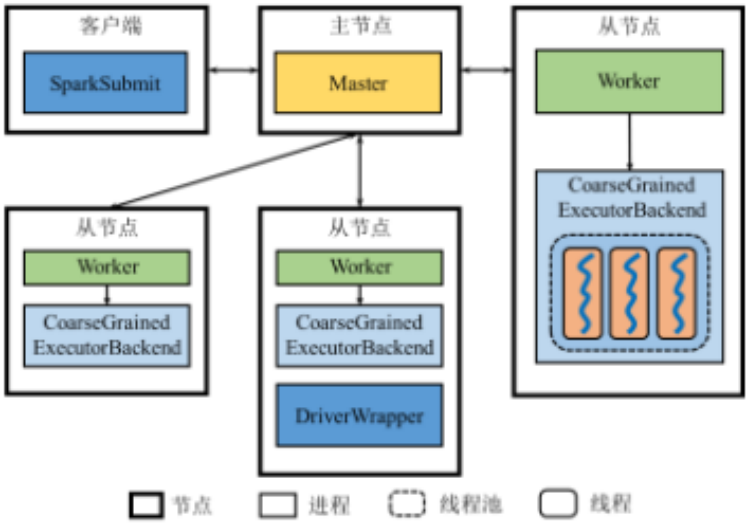

图 3.3: Spark Standalone Cluster Mode: Driver 集成在某个节点中

- Master+Worker=Cluster manager: 管理 Resource 的进程，监控 Worker 资源使用情况，分配资源
- CoarseGrainedExecutorBackend->Task: 管理 Task 线程

3.1.2 Yarn

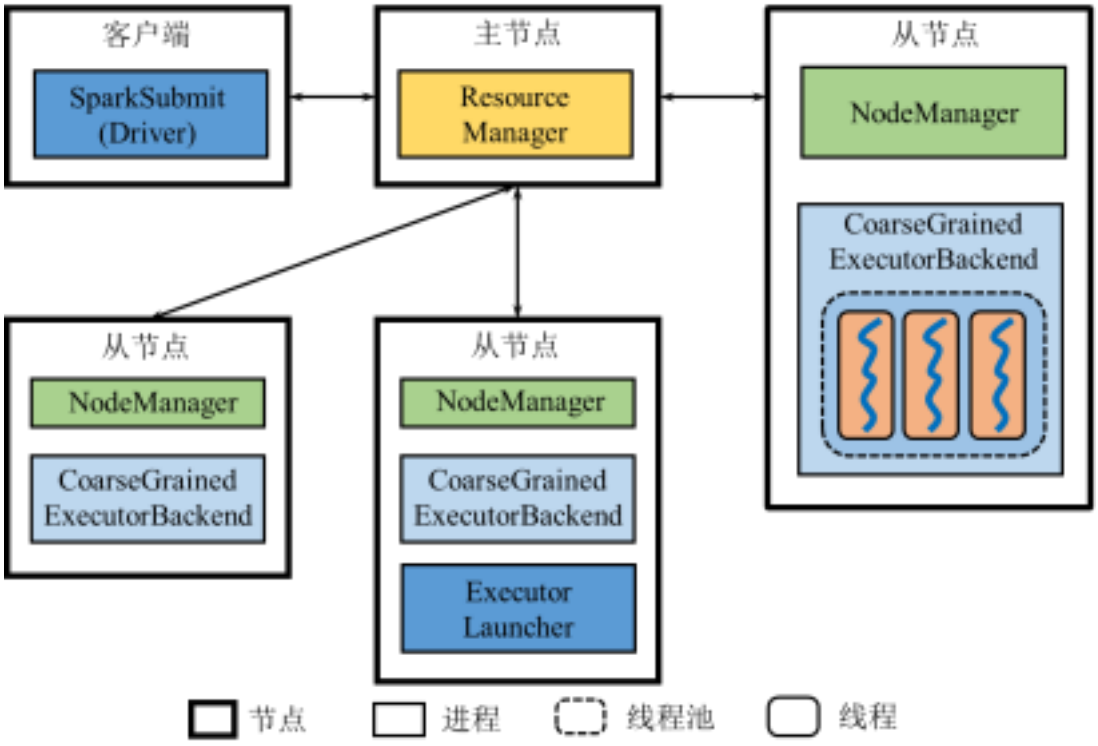

图 3.4: Spark Yarn Client Mode

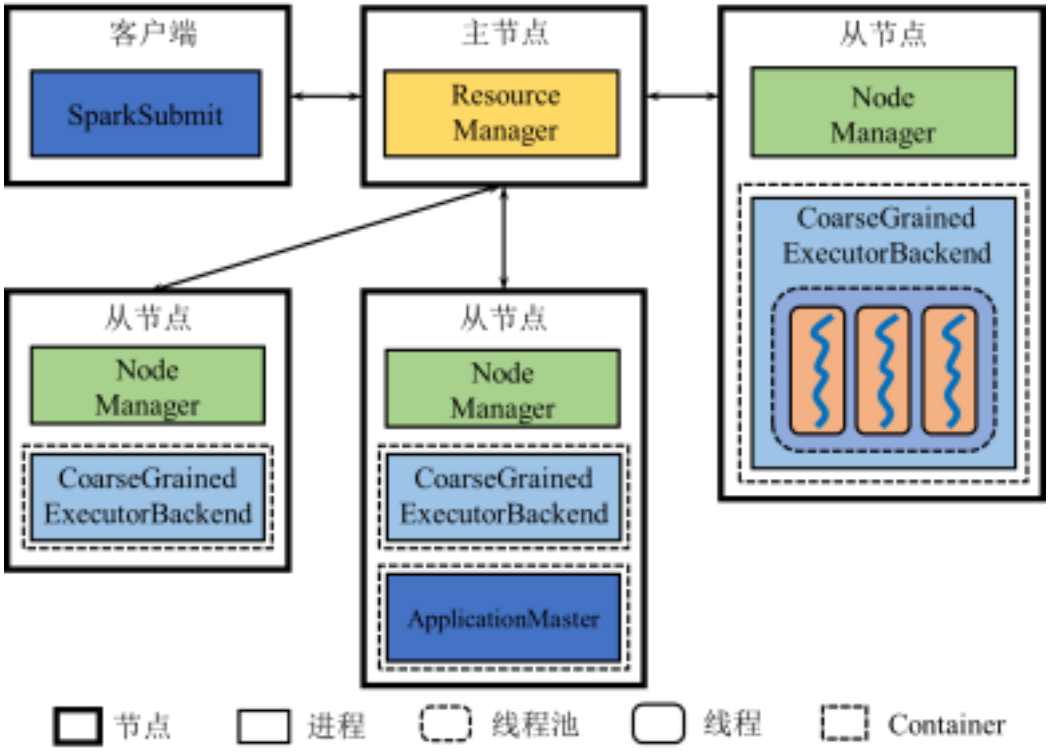

图 3.5: Spark Yarn Cluster Mode

3.2 工作原理

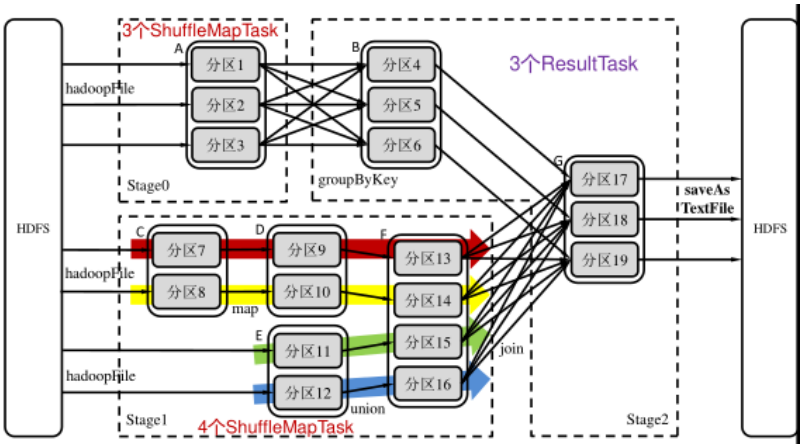

图 3.6: 宽依赖, 窄依赖, Task

- ShuffleMapStage (TaskSet): 中间结果
- ResultStage (TaskSet): 输出结果

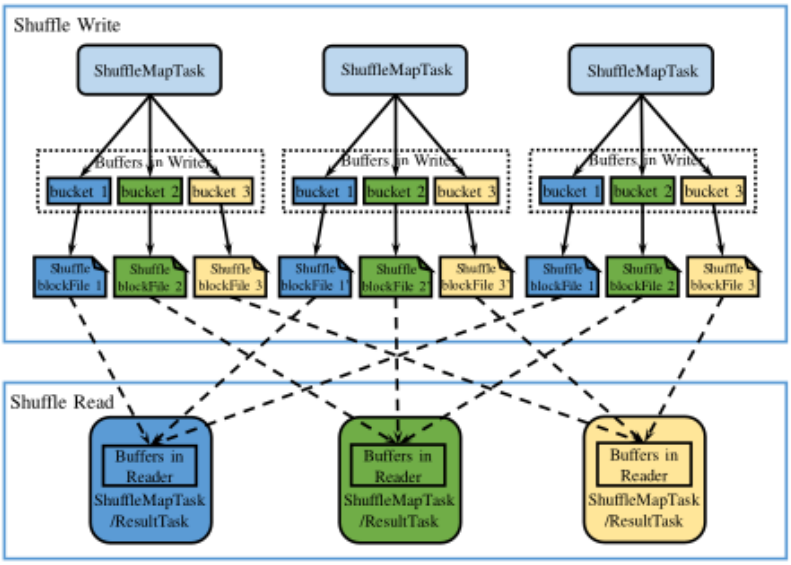

图 3.7: Shuffle

3.3 容错机制

3.3.1 Master 故障

ZooKeeper 配置多个 Master

3.4 RDD 持久化

支持多个缓存级别

- MEMORY_ONLY
- MEMORY_ONLY_SER (序列化)
- MEMORY_ONLY_2 (备份 2 台机器)
- MEMORY_AND_DISK
- MEMORY_AND_DISK_SER
- MEMORY_AND_DISK_2 (备份 2 台机器)
- DISK_ONLY

3.4.1 故障恢复

Lineage 机制: (重新计算丢失分区；重算过程在不同节点之间可以并行)

执行某个 partition 时，检查父 RDD 对应的 partition 是否存在:

- 存在: 即可执行当前 RDD 对应的操作
- 不存在: 窄依赖重构父 RDD 对应的 Partition, 宽依赖重构整个父亲 RDD

3.4.2 检查点机制

前述机制不足之处:

- Lineage 可能非常长
- RDD 持久化保存到集群内机器磁盘，不完全可靠

检查点机制将 RDD 写入外部可靠的 (本身具有容错机制) 分布式文件系统 (HDFS…)

4 Yarn

4.1 体系架构

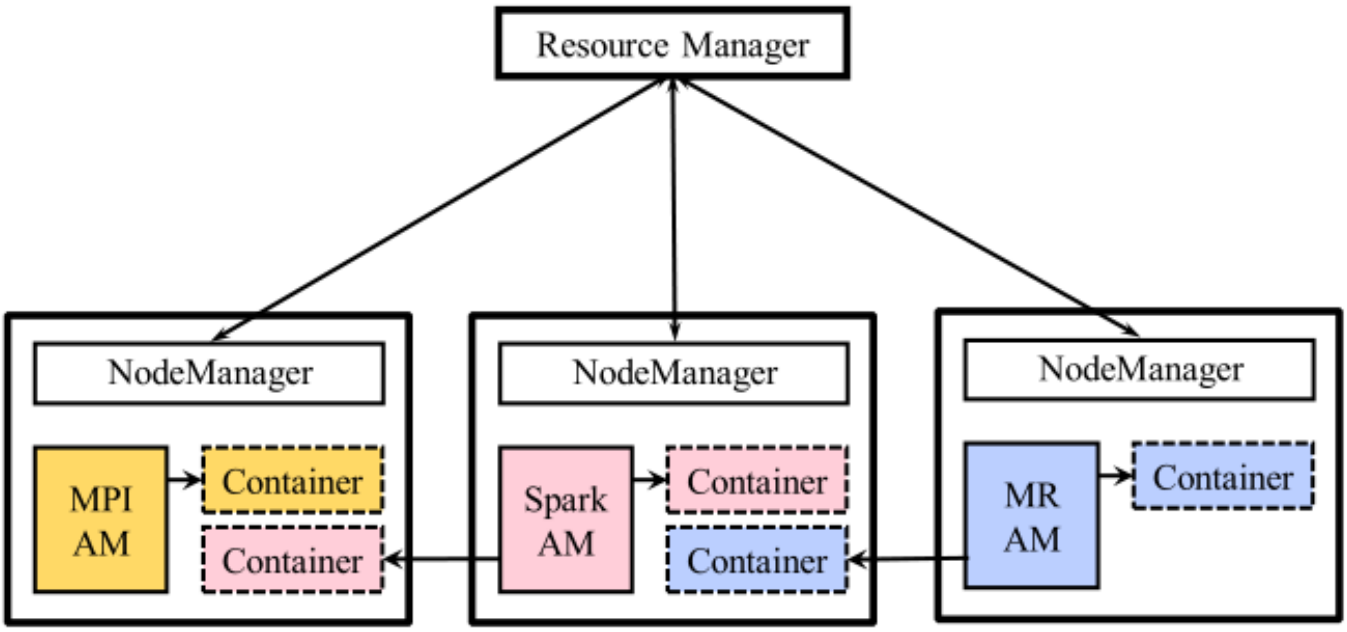

图 4.1: Yarn

- ResourceManager: 资源管理器：负责整个系统的资源管理和分配
  - 资源调度器 (Resource Scheduler)：分配 Container 并进行资源调度
  - 应用程序管理器 (Application Manager)：管理整个系统中运行的所有应用
    - 应用程序提交
    - 与调度器协商资源以启动 ApplicationMaster
    - 监控 ApplicationMaster 运行状态
- NodeManager: 节点管理器：负责每个节点资源和任务管理
  - 定时地向 RM 汇报本节点的资源使用情况和 Container 运行状态
  - 接受并处理来自 AM 的 Container 启动/停止等各种请求
- ApplicationMaster: 当用户基于 Yarn 平台提交一个框架应用，Yarn 均启动一个 AM 用于管理该应用
  - AM 与 RM 调度器协商以获取资源（以 Container 表示），将获取的资源进一步分配给作业内部的任务. 当 AM 向 RM 申请资源时，RM 向 AM 返回以 Container 表示的资源
  - AM 与 NM 通信以启动/停止任务，监控所有任务运行状态，并在任务发生故障时重新申请资源来重启任务
- Container: 资源的抽象表示，包含 CPU、内存等资源，是一个动态资源划分单位

4.2 工作原理

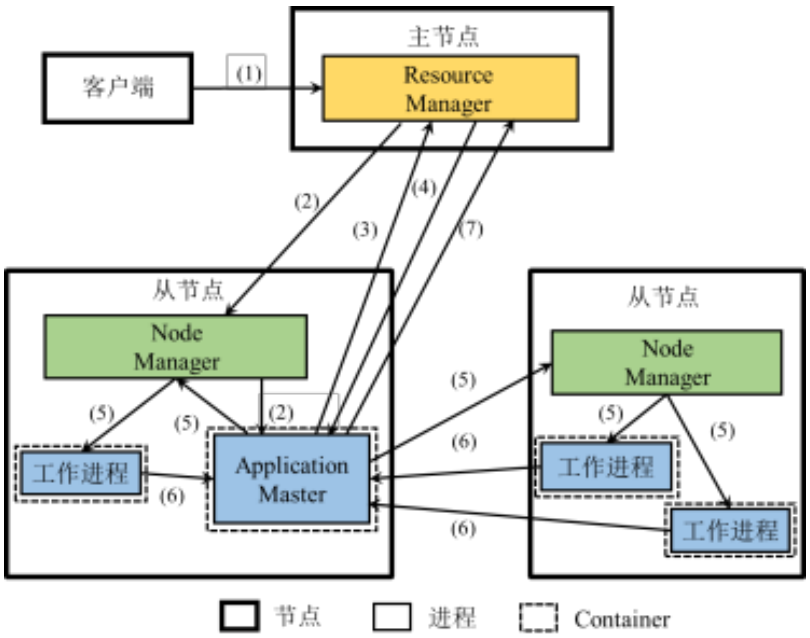

图 4.2: 执行流程

4.2.1 资源分配策略

- FIFO Schedular: 先来先得, 晚来阻塞
- Capacity Schedular: 按照资源占用划分多个 FIFO Schedular
- Fair Scheduler: 应用可以独占整个集群, 将空出的资源留给下一个应用

4.3 容错机制

- Resource Manager 故障: 进行故障恢复时需要从某一持久化存储系统中恢复状态信息，所有应用用将会重新执行；可以部署多个 Resource Manager 并通过 ZooKeeper 进行协调，从而保证 Resource Manager 的高可用性
- Node Manager 故障:
  - RM 认为 NM 所在节点上所有容器运行的任务都执行失败，并把执行失败的信息告诉 AM;
  - AM 将向 RM 重新申请资源运行这些任务，RM 将分配其它节点的 Container 执行这些任务.
  - 如果发生故障的 NM 进行恢复，那么它将向 RM 重新注册，重置本地的状态信息.

5 ZooKeeper

轻量级的分布式存储系统. 并不用于存储大量数据, 而是用于存储元数据或配置信息等.

5.1 数据模型

类似文件系统的树状结构, 节点称为 Znode. Znode 的类型有:

- PERSISTENT: 持久 Znode
- PERSISTENT_SEQUENTIAL: 持久顺序编号 Znode
- EPHEMERAL: 临时 Znode, 生命周期是客户端的 Session
- EPHEMERAL_SEQUENTIAL: 临时顺序编号 Znode

5.2 体系架构

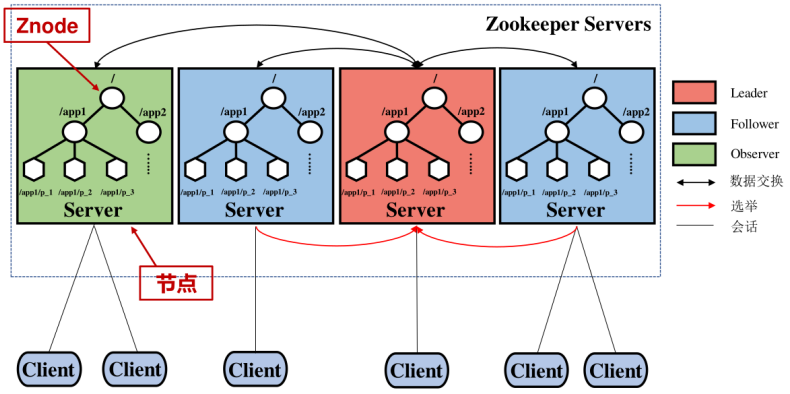

图 5.1: Zookeeper 架构图

5.3 工作原理

- 只读服务器转发写请求
- 只读服务器的数据不一定最新, 读请求前应手动先和主节点同步

5.4 容错机制

- Leader 故障: 重新选举
- Follower, Observer: 数据恢复

5.5 应用

5.5.1 命名服务

5.5.2 集群管理

- 状态监控: 每个节点创建一个临时 Znode, 监控节点 watch
- 选主: 例如 `[node_1, node_2, node_3]`, 每个节点 watch 前一个节点是否存在, 当前一个节点消失时, 自己成为主节点.

5.5.3 配置更新

所有节点 watch 某个 config znode

5.5.4 同步控制

双屏障机制:

- 进入屏障时，需要足够多的进程准备好
- 当所有进程执行完任务时，才可离开屏障

6 Storm

流处理系统

6.1 数据模型

流数据是一个无界的、连续的元组序列. 一个元组就是系统处理的一条记录, 每一条记录（元组）包含若干个字段.

6.2 计算模型

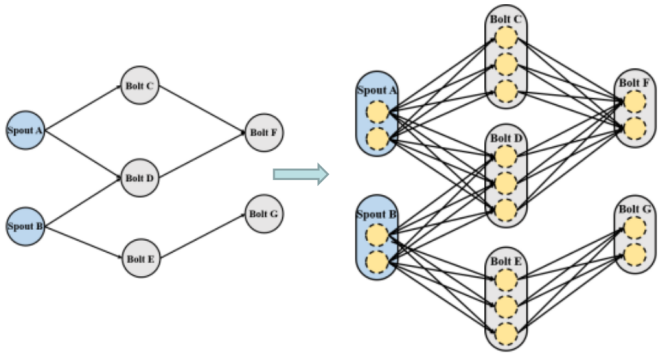

图 6.1: Storm 计算模型

由 Spouts 和 Bolts 组成的 DAG

- 顶点：Spout 或 Bolt, 数据处理逻辑. Spout/Bolt 物理上由若干个 task 来实现.
  - Spout：Stream 的源头，从外部数据源然后封装成 Tuple，发送给 Bolt
  - 描述 Streams 的转换过程，将处理后的 Tuple 作为新的 Streams 发送给其他 Bolt
- 边：Bolt 订阅的流数据, 数据流动的方向

6.3 体系架构

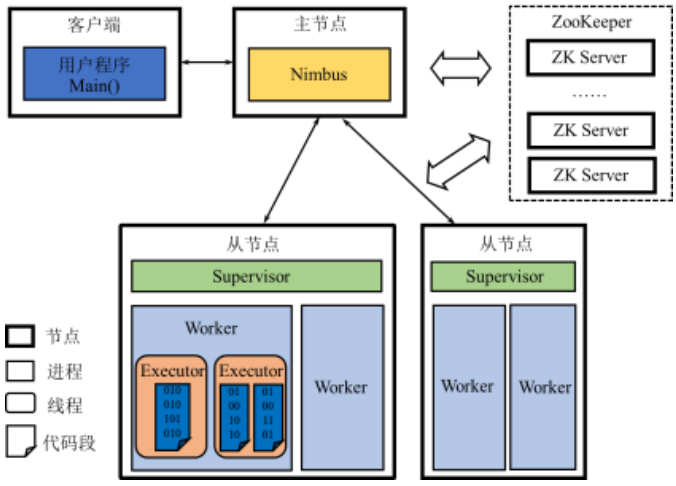

图 6.2: Storm 体系架构

- Nimbus：主节点运行的后台程序，负责分发代码、分配任务和监测故障
- Supervisor：从节点运行的后台程序
  - 负责监听所在机器的工作，根据 Nimbus 分配的任务来决定启动或停止 Worker 进程
  - 一个从节点上同时运行若干个 Worker 进程
- Zookeeper：负责 Nimbus 和 Supervisor 之间的所有协调工作
  - 若 Nimbus 进程或 Supervisor 进程意外终止，重启时也能读取、恢复之前的状态并继续工作
- Worker：Worker 进程内部运行一个或多个 Executor 线程，从而实际执行任务
  - Executor：产生于 worker 进程内部的线程，会执行同一个组件的一个或者多个 task
  - Task: 执行数据处理的代码实例（spout/bolt）

6.4 工作原理

6.4.1 流数据分组策略

- ShuffleGrouping：随机分组，随机分发 Stream 中的 Tuple，保证每个 Bolt 的 Task 接收 Tuple 数量大致一致
- FieldsGrouping：按照字段分组，保证相同字段的 Tuple 分配到同一个 Task 中
- AllGrouping：广播发送，每一个 Task 都会收到所有的 Tuple
- GlobalGrouping：全局分组，所有的 Tuple 都发送到同一个 Task 中
- DirectGrouping：直接分组，直接指定由某个 Task 来执行 Tuple 的处理

6.4.2 元组传递方式

一次一元组或一次一记录, 这种立即发送的消息传递机制有利于减少处理的延迟，从而满足实时性需求

6.5 容错机制

6.5.1 流计算系统容错语义

- At Most Once：消息可能会丢失
- At Least Once：消息不会丢失，可能会重复
- Exactly Once：消息不丢失，不重复

6.5.2 元组树

元组树: Spout 发出的 Tuple 及其衍生出来的 Tuple 抽象为一棵树.

STid(Spout-Tuple-id)：Spout 中发射 Tuple 时用户可以为其指定标识.

6.5.3 Acker

元组树中的元组非常多，并且往往并行地向 Acker 报告 Mid 和 STid

`<STid, ack_val>` 映射表

- 收到 Spout 发来消息时将相应 STid 的 ack_val 初始化为 0
- 无论 Acker 接收到上游组件还是下游组件报告的消息, 均将其中的 Mid 与映射表中相应 STid 的 ack_val 进行异或（XOR）操作
- 如果 Acker 在设定的时间范围内收到处于拓扑最末端的 Bolt 报告并且 ack_val 为 0，那么 Acker 会告诉相应的 Spout：STid 对应的元组树已经成功地处理完

6.5.4 消息重放

Spout 重新发送以 STid 为标识的元组, 但这种消息重放机制可能会导致消息的重复计算，达到的是至少一次的容错语义级别

7 Spark Streaming

根据用户指定的时间间隔进行切割 (离散化), 得到的每一小批数据的独立 RDD, DStream 维护一系列 RDD 信息.

7.1 体系架构

与 Spark 的区别:

- Driver: StreamingContext extends SparkContext
- Executor: 某些 Task 将负责从外部不断获取数据流

7.2 容错机制

- Cluster Manager 故障: 不考虑
- Executor(Worker) 故障
  - 不含 Receiver: 利用 RDD Lineage 进行恢复
  - 含有 Receiver: 利用日志进行恢复. Reciever 将数据存入本地和外部存储, 记录日志, 通知 Driver. Driver 记录元数据日志.
- Driver 故障: 利用 (元数据) 检查点进行恢复

7.2.1 端到端的容错语义

- 接收数据: at-least or exactly once

  取决于数据是使用 Receiver 或其它方式从数据源接收的

- 转换数据: exactly once

  接收到的数据是用 Dstream 和 RDD 做转换的

- 输出数据: at-least or exactly once

  取决于最终的转换结果被推出到外部系统如文件系统, 数据库, 仪表盘等

8 批流融合

8.1 Lambda 架构

平衡了重新计算高开销和需求低延迟的矛盾

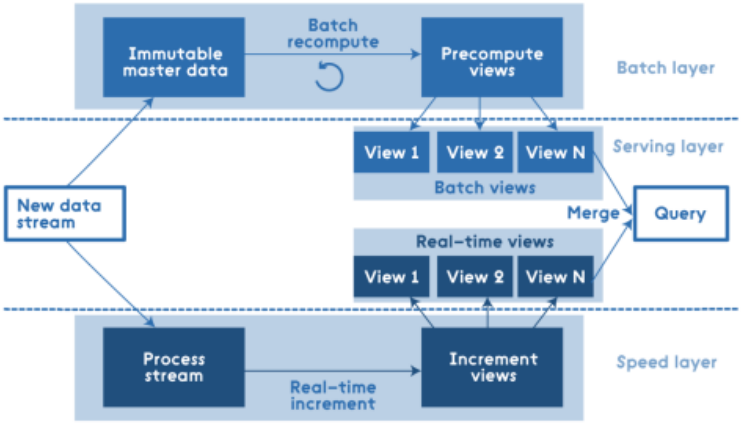

图 8.1: Lambda 架构

- Batch Layer
- Speed Layer: 实施处理新数据, 补偿批处理视图
- Serving Layer (数据库)

缺点:

- 开发复杂：需要将所有的算法实现两次，批处理系统和实时系统分开编程，还要求查询得到的是两个系统结果的合并
- 运维复杂, 同时维护两套执行引擎

解决方案:

1. 批流系统分开编程变为统一编程: Google Dataflow, Apache Beam
2. 如何将两套执行引擎变为一体化的批流融合执行引擎并支持统一编程: Spark Structured Streaming, Apache Flink

8.2 批处理与流计算的统一性

8.2.1 有界 vs. 无界数据集

8.2.2 窗口操作

把数据集切分为有限的数据偏以便针对该数据片进行处理

- 滑动窗口: range > slide
- 滚动窗口: range = slide
- 跳跃窗口: range < slide

8.2.3 时间域

- 事件时间 Event Time：该事件实际发生的时间

  当该事件发生时，其所在系统（可能是传感器等数据源）的当前时间

- 处理时间 Processing Time：一个事件在数据处理的过程中被数据处理系统观察到的时间

  当该事件被数据处理系统处理时，数据处理系统的当前时间

- 一个记录的事件时间是永远不变的，但是处理时间随着该记录在系统中被各个节点处理时而持续变化

- 水位线 (watermark): 水位线是由系统根据预定义或用户指定的启发式规则生成的, 因此“认定”实际上只是一种猜测

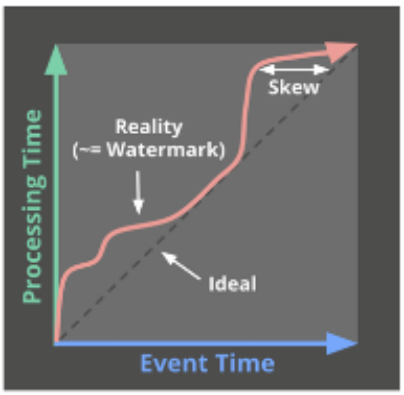

图 8.2: Watermark

8.3 Dataflow 统一编程模型

8.3.1 操作描述 (What)

- ParDo: 该操作对每个键值对都执行相同的处理，获得 0 或多个输出键值对
- GroupByKey: 用来按键把元素重新分组

8.3.2 窗口定义 (Where)

例如事件时间窗口

8.3.3 触发器 (When)

在某一**处理时间**决定处理窗口的聚合结果并输出, 如果水位线过慢可以提前触发

8.3.4 乱序修正 (How)

水位线过快, 数据迟到. 如何管理当前的窗口内容?

- 抛弃 (Discarding): 触发器一旦触发后，窗口内容即被抛弃，之后窗口计算的结果和之前的结果不存在任何相关性
- 累积 (Accumulating): 触发器触发后，窗口内容进行持久化，而新得到的结果成为对已输出结果的一个修正版本
- 累积和撤回（Accumulating & Retracting): 触发器触发后，不仅将窗口内容持久化，还需记录已经输出的结果。当窗口将来再触发时，先撤回已输出的结果，然后输出新得到的结果

8.4 一体化执行引擎

- 以批处理为核心 (秒级延迟): 将无界数据集划分为小批量数据，不断地启动短作业来处理这些小批量数据. 先启动的作业必须执行完，才启动新作业.
- 以流计算为核心 (毫秒级延迟): 启动一个长期运行的作业. 对于有界数据集来说，相当于系统接收一定量的记录之后就不再接收新的记录了

9 Flink

9.1 数据模型

- DataStream: 不间断的、无界的连续记录序列

9.1.1 DataStream 操作算子

一系列的变换操作构成一张有向无环图，即描述计算过程的 DAG

- 数据源（DataSource）
- 转换（Transformation）
- 数据池（DataSink）

9.1.2 计算模型

Flink 系统中的一个应用对应一个 DAG. Flink 中使用迭代算子实现迭代. 但流计算中的迭代会把每一次迭代结果继续传播, 与批处理中语义不同.

在流式迭代计算中，通常每一轮迭代计算的部分结果作为输出向后传递，而另一部分结果作为下一轮迭代计算的输入，并且迭代过程会一直进行下去.

在批式迭代计算中，每一轮迭代计算的全部结果通常都是下一轮迭代计算的输入，直到迭代过程在满足收敛条件时停止迭代.

9.2 体系架构

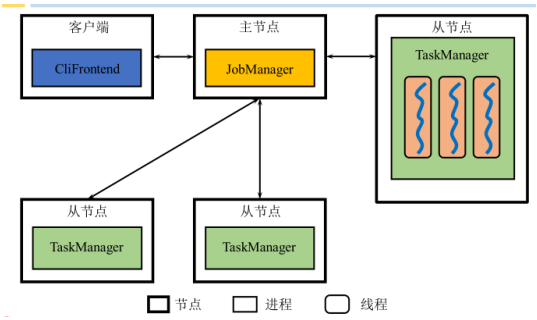

图 9.1: Flink

- Client: 将用户编写的 DataStream 程序翻译为逻辑执行图并进行优化，并将优化后的逻辑执行图提交到 JobManager
- JobManager：作业管理器. 根据逻辑执行图产生物理执行图，负责协调系统的作业执行，包括任务调度，协调检查点和故障恢复等.
- TaskManager：任务管理器. 用来执行 JobManager 分配的任务，并且负责读取数据、缓存数据以及与其它 TaskManager 进行数据传输.

客户端可选 Attached/Detached.

9.2.1 Standalone

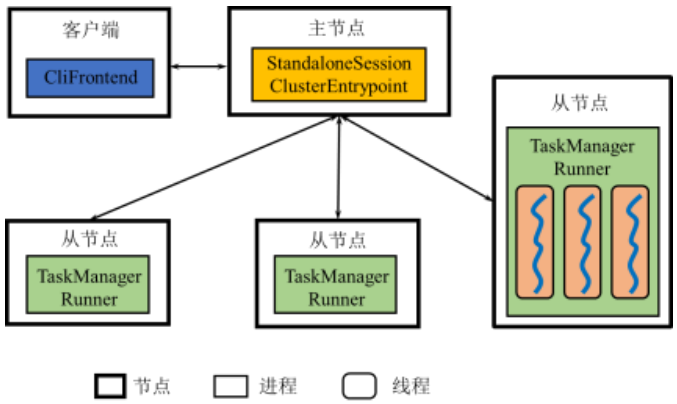

图 9.2: Flink standalone

9.2.2 Yarn

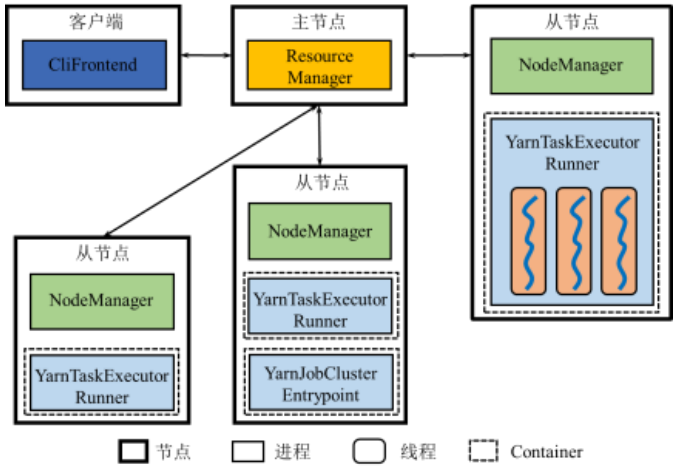

图 9.3: Flink yarn

9.3 工作原理

9.3.1 逻辑执行图优化

Chaining: 合并窄依赖的算子

9.3.2 物理执行图

根据任务槽 (TaskSlot) 的容量，尽可能将存在数据传输关系的算子实例放在同一个任务槽，保持数据传输的本地性

9.3.3 非迭代任务间的数据传输

Pipeline: 上游的 Task 将数据存在 buffer 中，一旦 buffer 满了或者超时，就向下游 Task 发送 (非阻塞, 也因此不存在 Lineage)

9.3.4 迭代任务内部的数据传输

- 迭代前端（Iteration Source）和迭代末端（Iteration Sink）两类特殊的任务
- 两类任务成对处于同一个 TaskManager，迭代末端任务的输出可以再次作为迭代前端任务的输入

9.4 容错机制

9.4.1 JobManager 故障

重新启动或 JobManager 高可用化

9.4.2 TaskManager 故障

9.4.2.1 状态管理

利用系统题库的系统定义的特殊的数据结构 (状态, `ValueState<T>`, `ListState<T>`, `ReducingState<T>` 保存有状态算子的状态 (计算结果).

- 算子级别的容错

  运行时保存其状态，在发生故障时重置状态，并继续处理结果尚未保存到状态当中的记录

- DAG 级别的容错

  如果我们可以在“同一时刻”将所有算子的状态保存起来形成检查点，一旦出现故障则所有算子都根据检查点重置状态，并处理尚未保存到检查点中的记录. 要求所有节点的物理时钟绝对同步, 但不可能.

9.4.2.2 非迭代 Task

- 在某一时刻，流计算系统所处理的记录可以分为三种类型
  - 已经处理完毕的记录，即所有算子都已经处理了这些记录
  - 正在处理的记录，即部分算子处理了这些记录
  - 尚未处理的记录，即没有算子处理过这些记录
- 虽然绝对同步的时钟是不存在的，但是同一时刻保存所有算子状态到检查点的目的是**区分第一种记录与后两种记录**

异步屏障快照: Chandy-Lamport 算法, 异步屏障快照 (数据流中插入屏障, 每个算子屏障对齐时保存快照)

当发生故障时, Flink 选择最近完整的检查点 n，将系统中每个算子的状态重置为检查点中保存的状态. 然后从数据源重新读取属于屏障 n 之后的记录 (这要求数据源具备一定的记忆功能). 能够满足 **exactly once** 的容错语义

9.4.2.3 迭代 Task

仅有屏障是无法区分的, 根据 Chandy-Lamport 算法, 反馈环路中的所有记录需要以日志形式保存起来.

10 Giraph

BSP: `loop(superstep: compute() -> sendMessage() ->` 栅栏同步`)`