Lecture 16 流计算模型

- 流计算概念
- 流计算模式
- 流计算模型

■ 流计算概念

1998年通信领域的美国学者Monika R. Henziger 将流数据定义为“只能以事先规定好的顺序被读取一次的数据的一个序列”。数据流可采用如下的形式化描述：

考虑一个向量α，其属性域为[1 ... n] (n为秩)，则向量α在时间t的状态可表示为

$\alpha(t) = <\alpha 1(t), \ldots \alpha i\ (t), \ldots \alpha n\ (t)>,\ i = 1, 2, \ldots, n$

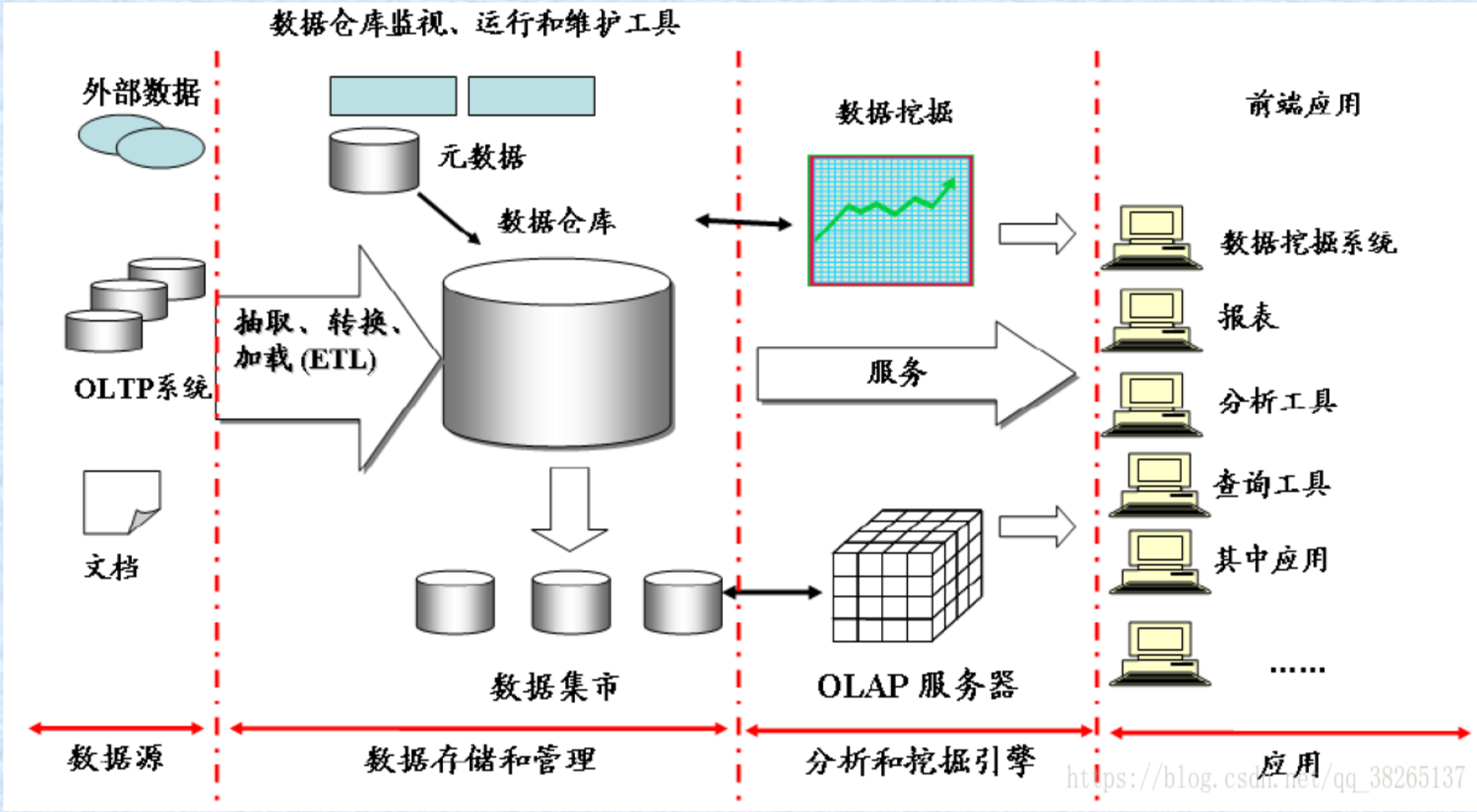
可设定在时刻s，α是0向量，即对于所有属性i，αi (s) = 0。

向量值的改变是基于时间变量的线性叠加，即时刻t各个分量的更新是基于（t-1）时刻以二元组流的形式出现的。即t时刻第i个更新为(i, ct)，意味着

$\alpha i\ (t) = \alpha i\ (t-1) + ct$

针对上述流数据的计算模式称为 **流计算（Stream Computing）**。

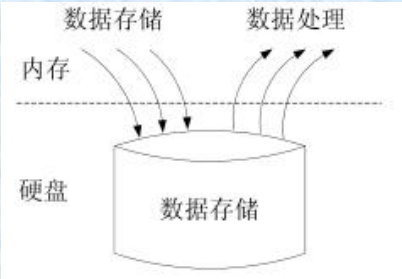
静态数据处理过程



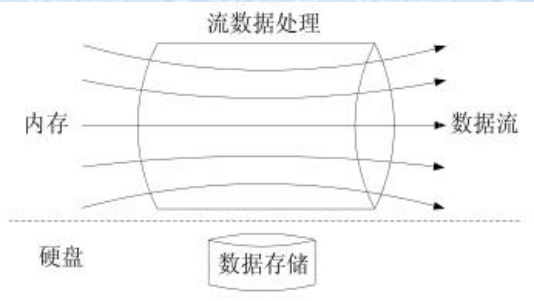
MapReduce模型 vs. 流计算模型

MapReduce批处理（batch processing）模型是先将数据存储于文件系统或数据库，然后对存储系统中的静态数据进行处理计算，这一步骤并不是实时在线的，因此又被称为离线批处理模式。

流计算(stream computing)则是在数据到达同时即进行计算处理，计算结果也实时输出，原始输入数据可能保留，也可能丢弃。



批处理计算



流计算

图 15-1　批处理模式 vs. 流计算模式

流计算系统模型

分布式系统中常用有向非循环图（DAG, Directed Acyclic Graph）来表征计算流程或计算模型。如下图就表示了分布式系统中的链式任务组合，图中的不同颜色节点表示不同阶段的计算任务（或计算对象），而单向箭头则表示了计算步骤的顺序和前后依赖关系。

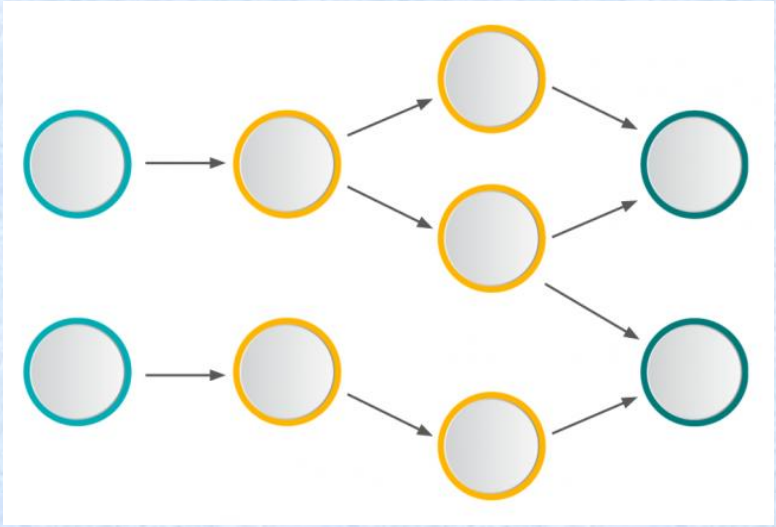


图 15-2 有向非循环向图（DAG）

■ 流计算模式

◘ Native Stream Processing System

基于数据读入顺序逐条进行处理，每一条数据到达即可得到即时处理（假设系统没有过载），简便易行，系统响应性好。但系统吞吐率（throughput）低，容错成本高和容易负载不均衡。

◘ Micro-batch Stream Processing System

将数据流先作预处理，打包成含多条数据的batch（批次）再传送给系统处理，系统吞吐率高，但延迟时间长。

Native Stream Processing System

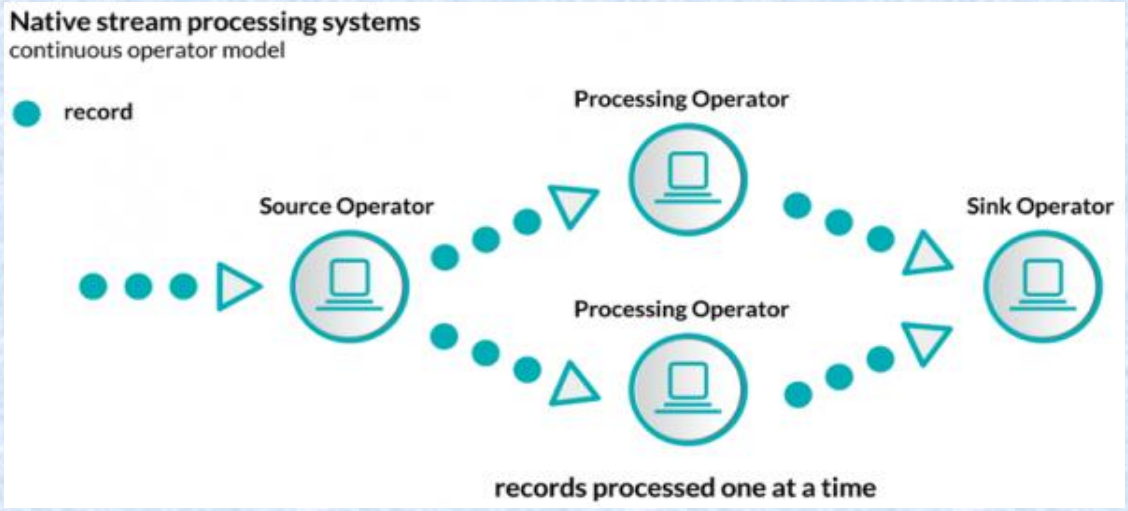


图 15-3　Native Stream Processing System

Micro-batch Stream Processing System

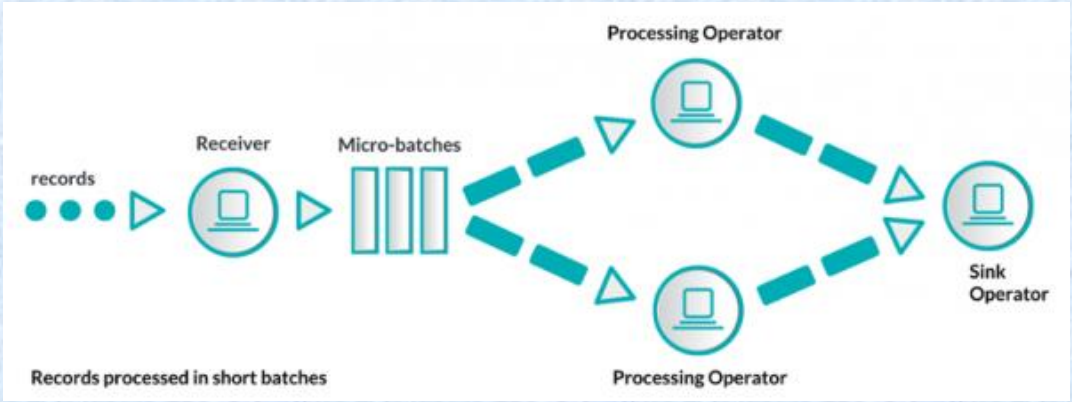
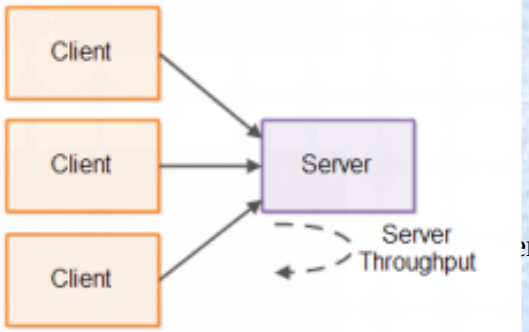


图 15-4　Micro-batch Stream Processing System

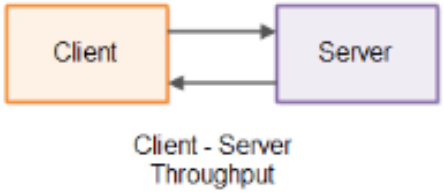
流计算性能参数

系统吞吐率（system throughput）：指单位时间内系统处理的数据量或完成的任务数。

对于Client/Server系统而言，服务器端的吞吐率是指服务器在单位时间内对所有的客户端完成的任务数；客户端的吞吐率则是指对单个客户而言服务器在单位时间内完成的该客户提交的任务数目。在讨论系统吞吐率时，我们一般是指的服务器端的吞吐率。



erver系统



系统响应时延（response delay）：基于客户端计算的，向服务器端提交一个任务到计算结果返回之间的时间间隔。

假设客户端一个任务完成的时间可分为如图三部分：去时传输时间、返回传输时间、服务器处理时间

```
delay time = network latency+server latency+network latency
           = 2 * network latency + server latency
```

其中，network latency是网络传输时间以L_n表示；server latency是服务器处理一条数据所需时间，以L_s表示。

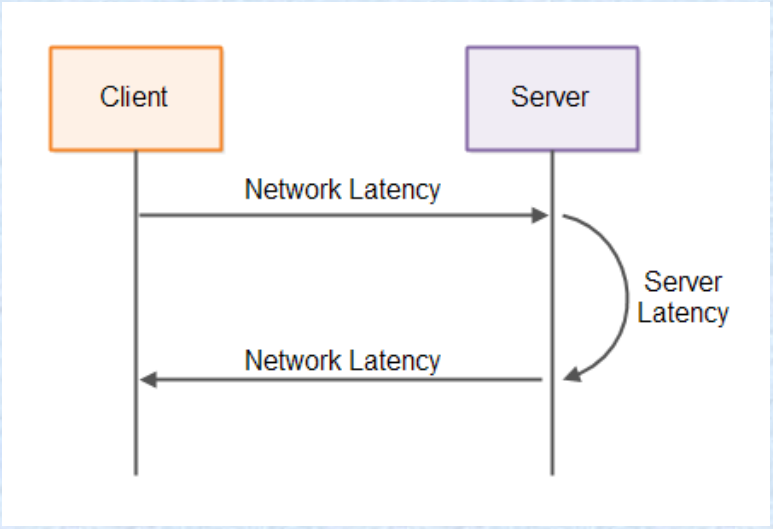


图 15-6　Client/Server 系统响应时间

两种流计算模式比较

对于Native Stream Processing System而言：

客户端系统延迟 delay time = $2L_n + L_s$

服务器端系统吞吐率 throughput = 1/(delay time) = $1/(2L_n + L_s)$

如果我们采用一次把10条数据打成一个包（batch）发送处理的方式，网络传输时间不变，仍然为2 * network latency，但服务器处理时间变为10 * server latency，因为需要处理10条数据。

对于Micro-batch Processing System而言：

客户端系统延迟 delay time = $2L_n + 10L_s$

服务器端系统吞吐率 throughput = 10/(delay time) = $10/(2L_n + 10L_s)$

= $1/(0.2L_n + L_s)$

比较两种模式的delay time和throughput可知，只要$L_n > 0$和$L_s > 0$，则有：

$$2L_n + L_s < 2L_n + 10L_s$$

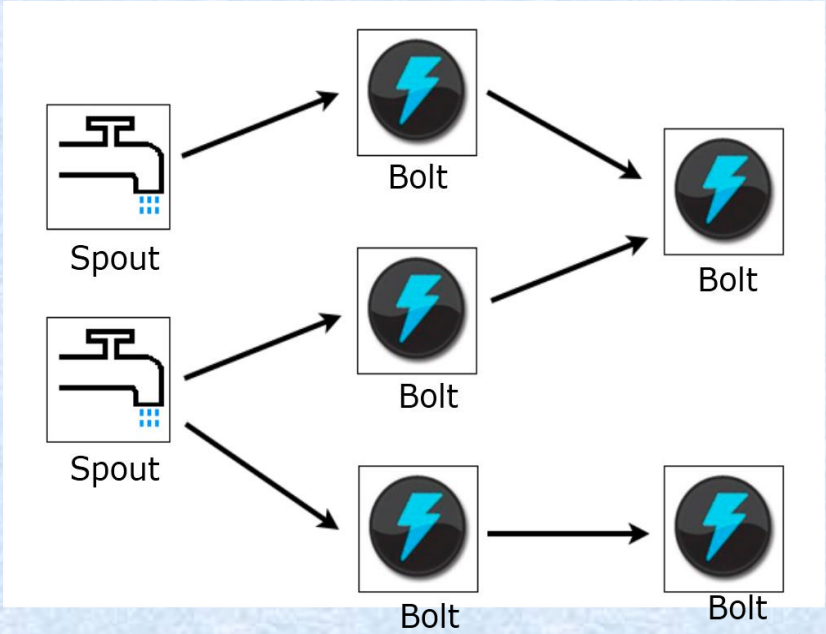
$$1/(2L_n + L_s) < 1/(0.2L_n + L_s)$$

■ 三种流计算模型

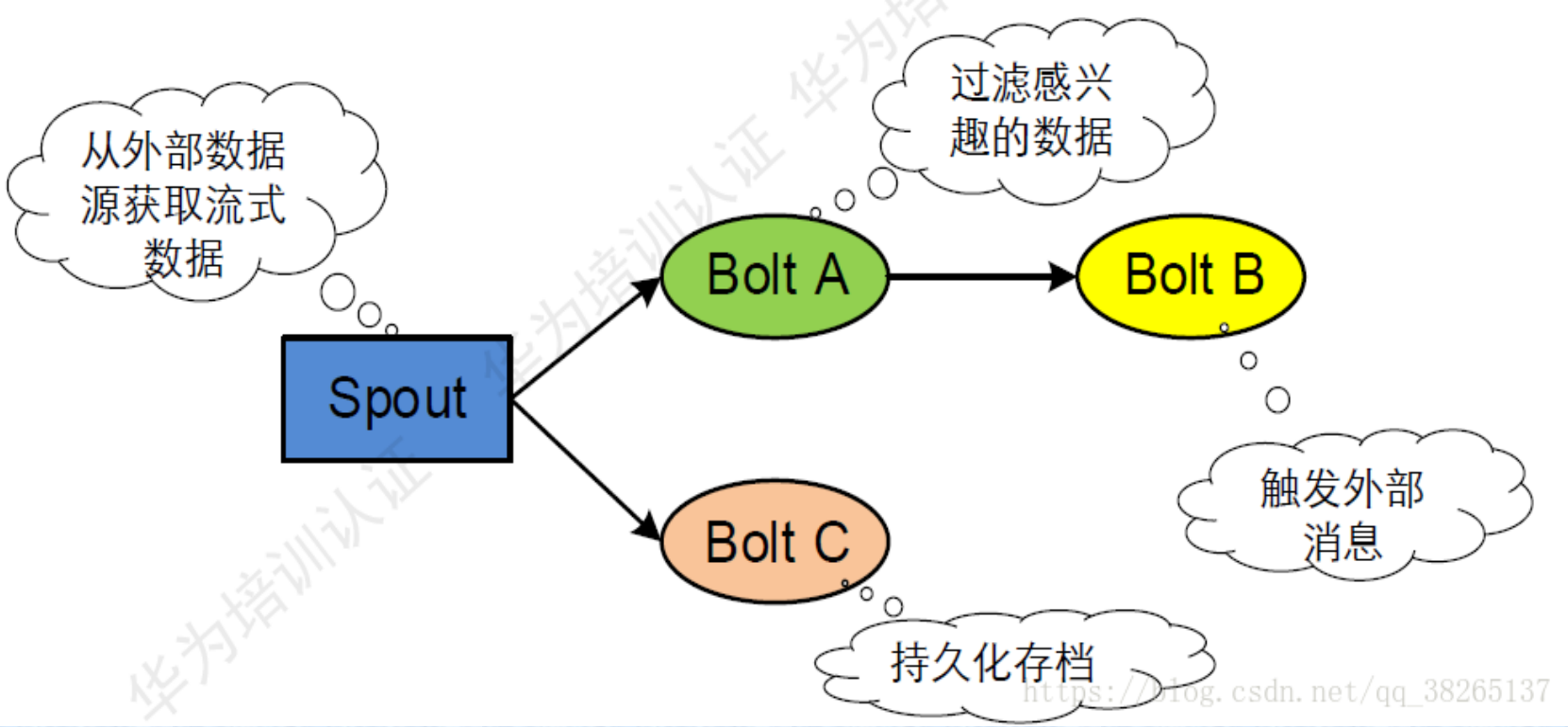
➢ Storm的Topology模型

Storm是一种Native Stream Processing System，即对流数据的处理是基于每条数据进行，其并行计算是基于由Spout（数据源）和Bolt（处理单元）组成的有向拓扑图Topology来实现。

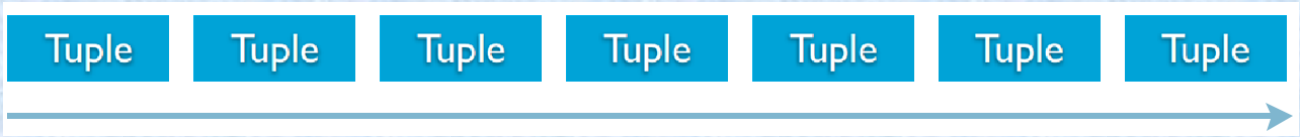
Topology：定义了并行计算的逻辑模型（或者称抽象模型），也即从功能和架构的角度设计了计算的步骤和流程。



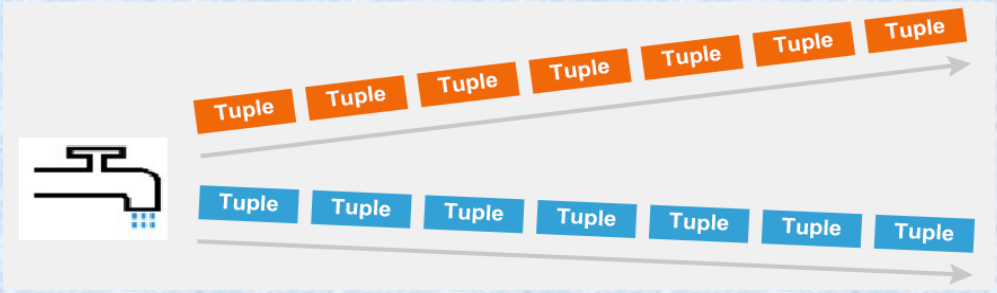
Topology组成



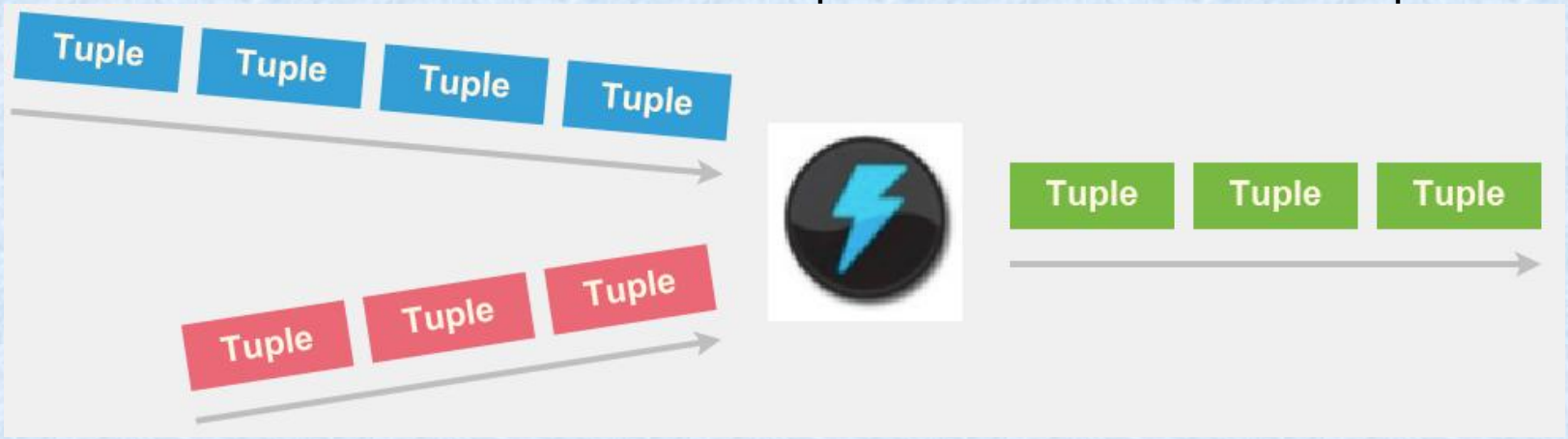
Tuple： 基本数据单元，可看作一组各种类型的值域组成的多元组。



Spout： 数据源单元，负责将输入数据流转换成一个个Tuple, 发送给Bolt处理。



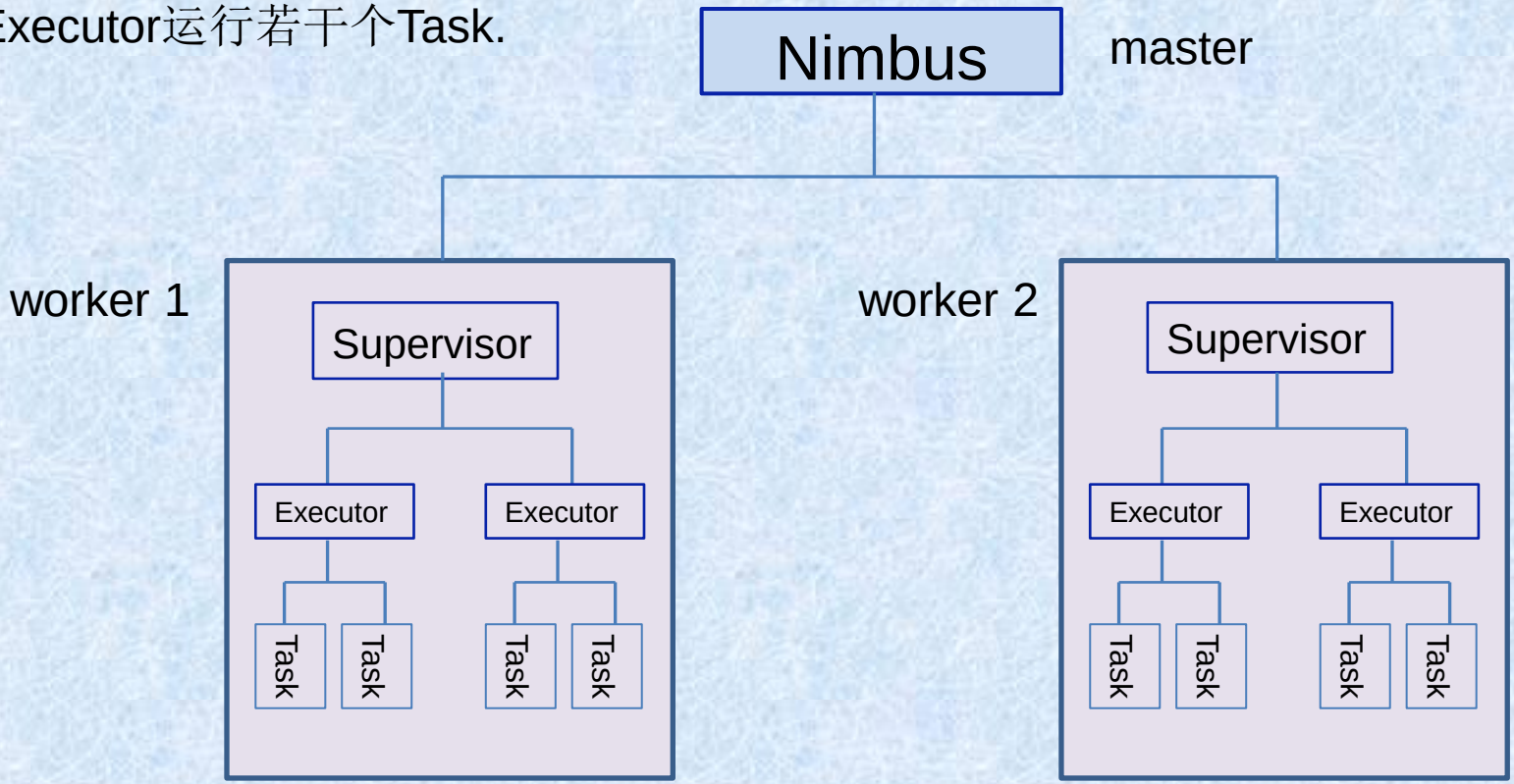
Bolt： 处理单元，负责读取上游传来的Tuple，向下游发送处理后的Tuple。



主从架构 Nimbus(master) / Supervisor(worker)

实现上述逻辑模型的物理架构：

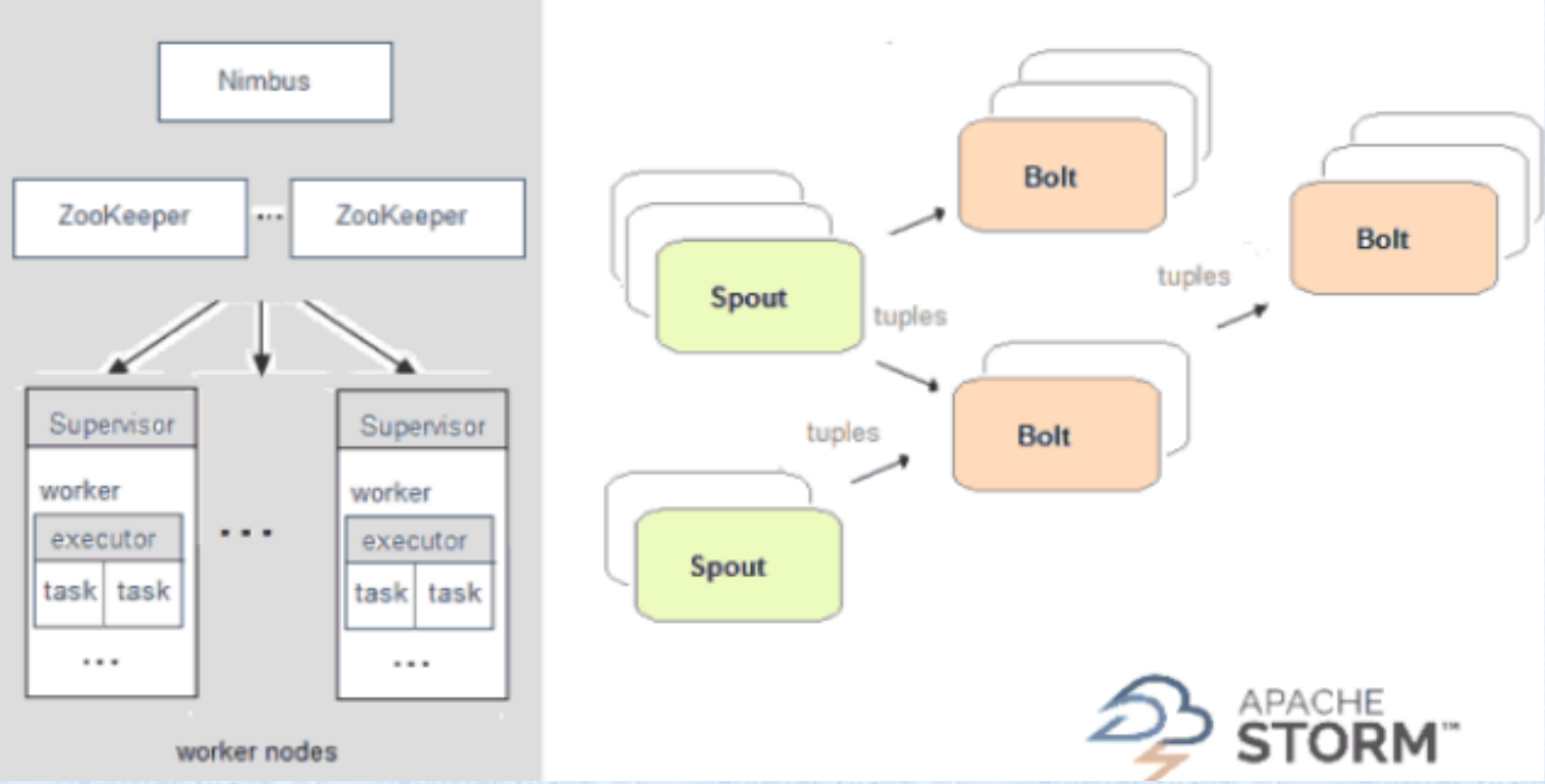
- 每一台机器(Worker)上运行一个Supervisor;
- 每个Supervisor管理若干个Executor ；
- 每个Executor运行若干个Task.



逻辑架构： Topology / Tuple / Spout / Bolt

物理架构： Nimbus / Supervisor / Executor / Task

如何完成“逻辑架构 ⟹ 物理架构”的映射（调度分发）？



- 当一个Storm作业被提交时，同时需要提交预先设计的逻辑Topology
- Topology里的Spout和Bolt的功能是靠worker节点上的Task来实现
- 一个Spout或Bolt的任务需要不同worker上的多个Task来并行完成

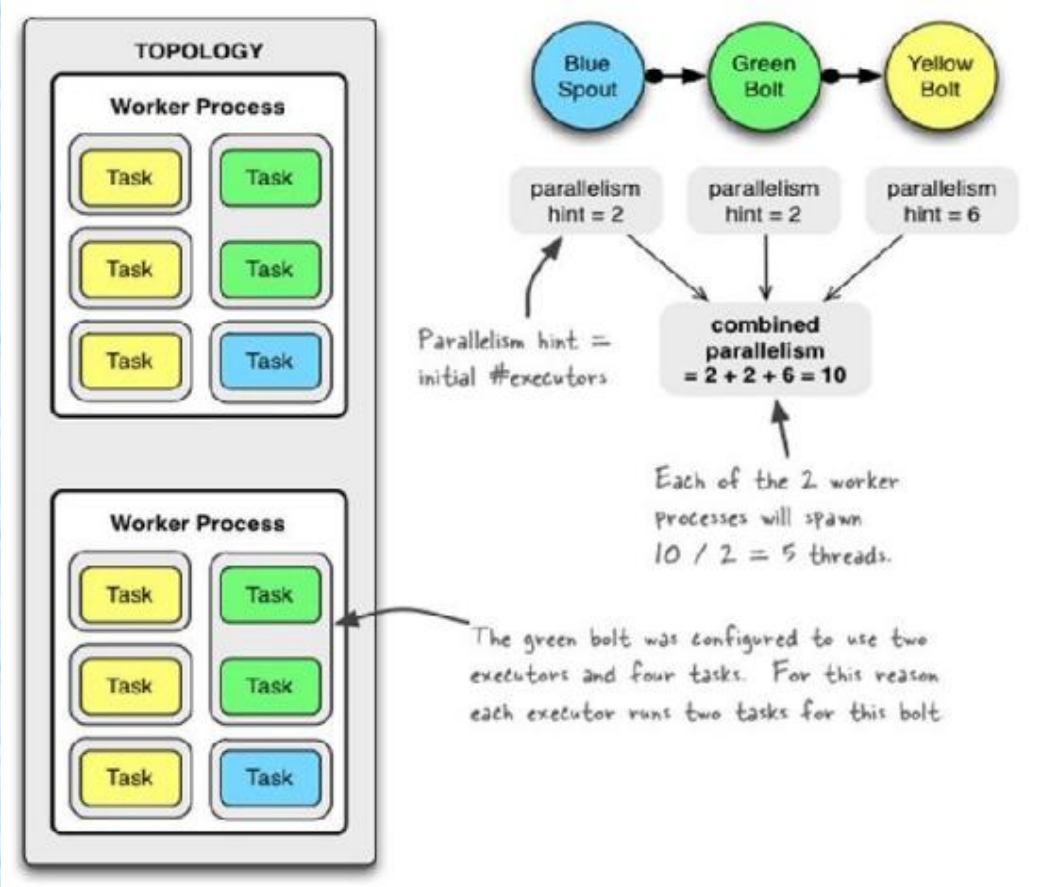
对于左图的1 Spout + 2 Bolts的topology，预先设计为：

Spout 并行度　= 2 executors
绿色Bolt并行度 = 2 executors
黄色Bolt并行度 = 6 executors

由此可计算出各个组件所需要的task数目（每个Green Bolt executor 并发2个task）：

#task = 2 + 2*2 + 6 = 12

这12个线程分配到2个Worker，每个Worker需运行12/2 = 6 个线程，分属于5个executor。



➢ Spark的DStream模型

Spark 流计算的核心概念是 Discretized Stream（DStream），DStream 由一组 RDD 组成，每个 RDD 都包含了规定时间段（可设置）流入的数据。Spark流计算可以基于单个RDD处理、也可基于时间window（包含多个RDD）进行处理。

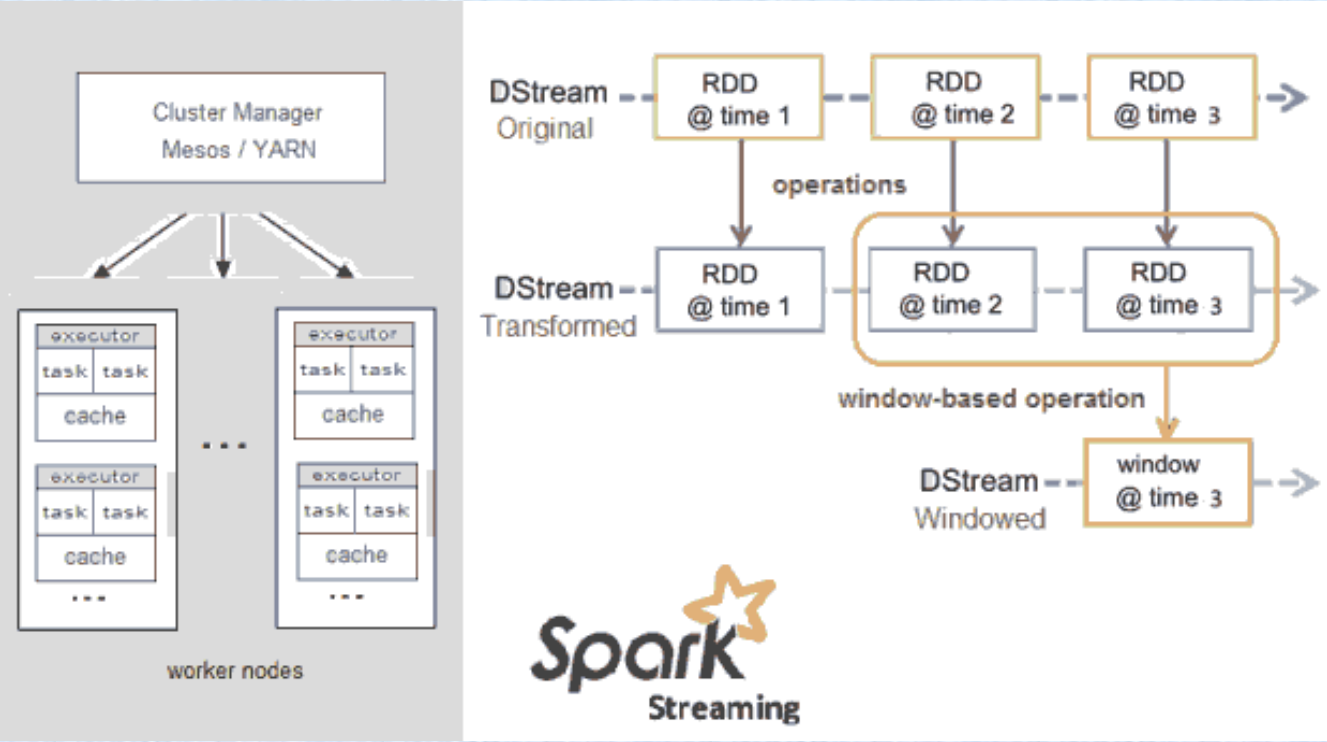


图 15-11　Spark 的并行计算模型

Spark的计算程序分为Driver（运行在Master节点上，也有一种模式运行在某一Worker节点上）和Executor（运行在Worker节点上）两部分：

- Driver 负责把应用程序的计算任务转化成有向非循环图（DAG）
- Executor 则负责完成worker节点上的计算和数据存储
- 在每个worker上， Executor针对一个个分发给它的数据partition再生成一个个Task线程，完成并行计算

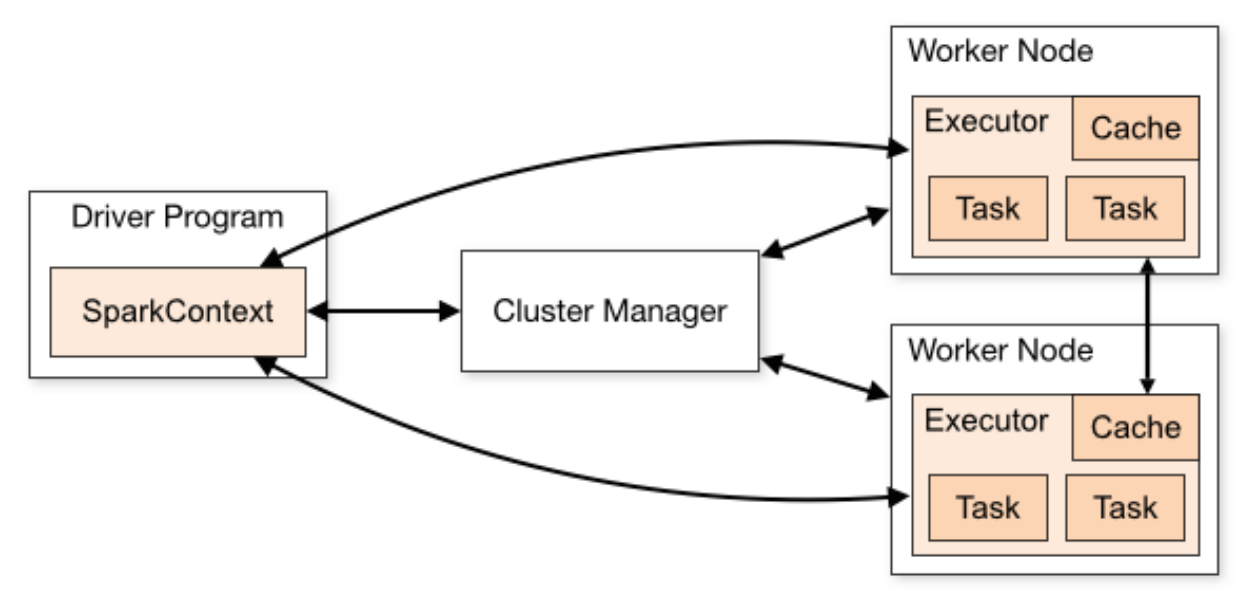
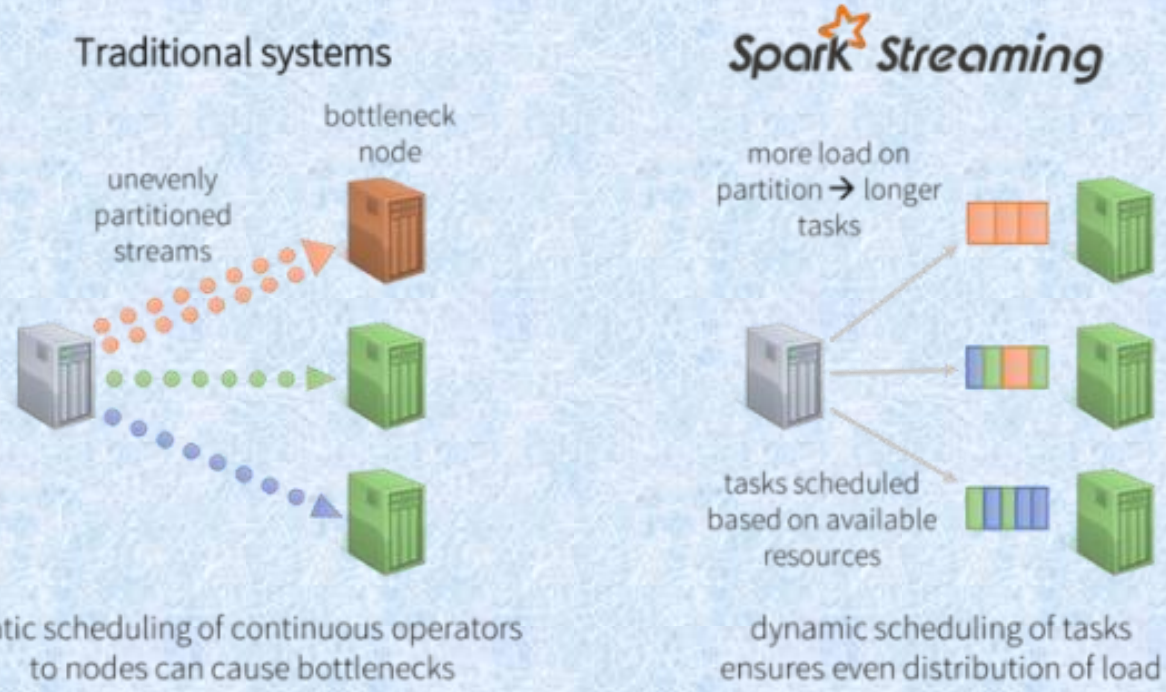


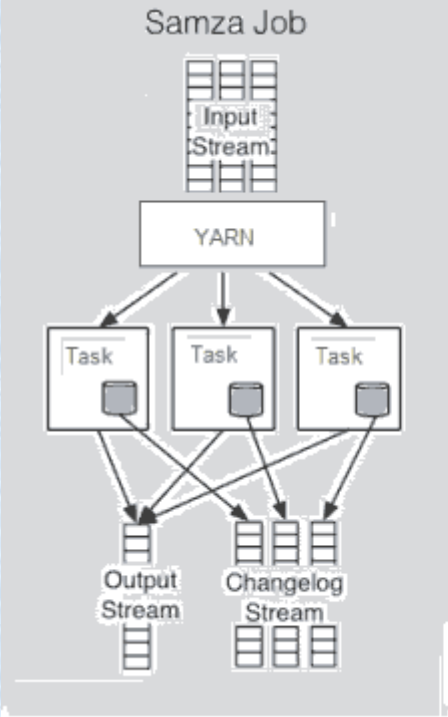
图 15-12　Spark 计算体系

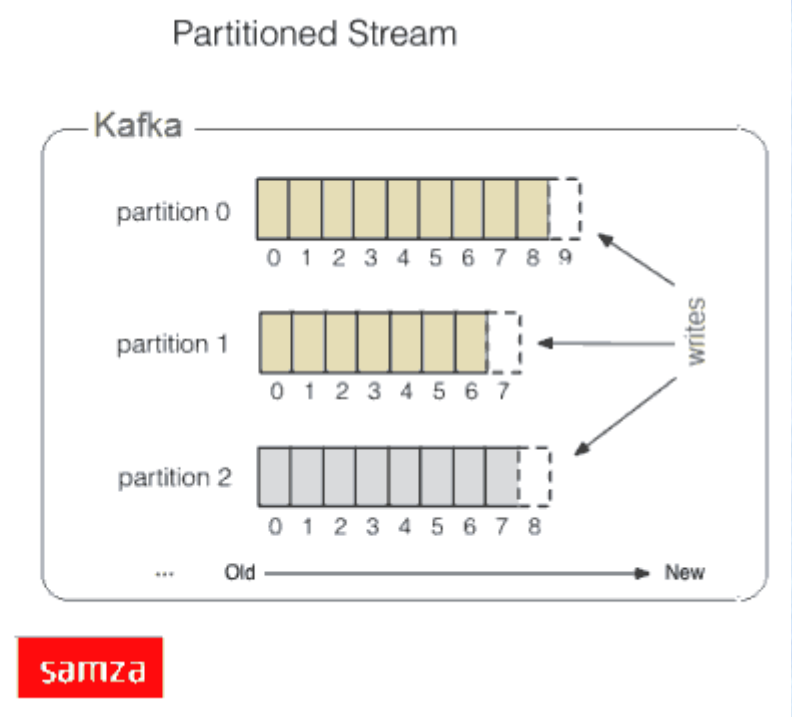
由于Spark将RDD划分为更小尺度的分区，因此可对资源进行细粒度分配。例如，输入DStream需要按键值来进行处理，传统处理系统会把属于一个RDD的所有分区分配到一个worker（图左边所示），如果一个RDD的计算量比别的RDD大许多，就会造成该节点成为性能瓶颈。而在Spark Streaming中，属于一个RDD的分区会根据节点荷载状态动态地平衡分配到不同节点上（图右边），一些节点会处理数量少但耗时长tasks，另一些节点处理数量多但耗时短的tasks，使得整个系统负载更均衡。



➢ Samza的Partitioned Stream模型

由LinkedIn 开源的一个分布式流处理系统，与之配合使用的是Apache开源分布式消息处理系统Kafka，也是一种基于逐条消息处理的Native Stream Processing System，强调的是对数据流的低延迟快速处理，但与Storm基于Topology的计算模式不同，Samza的并行计算是基于Kafka提供的分区数据流（partitioned stream）。





➢ Flink：一个分布式流处理开源框架

2008 年，Flink 的前身已经是柏林理工大学一个研究性项目

2014 被 Apache 孵化器所接受，然后迅速地成为了 ASF（Apache Software Foundation）的顶级项目之一

Flink特性

1： 即使数据源是无序的或者晚到达的数据，也能保持结果准确性

2： 有状态并且容错，可以无缝地从失败中恢复，并可以保持 **exactly-once**

3： 大规模分布式

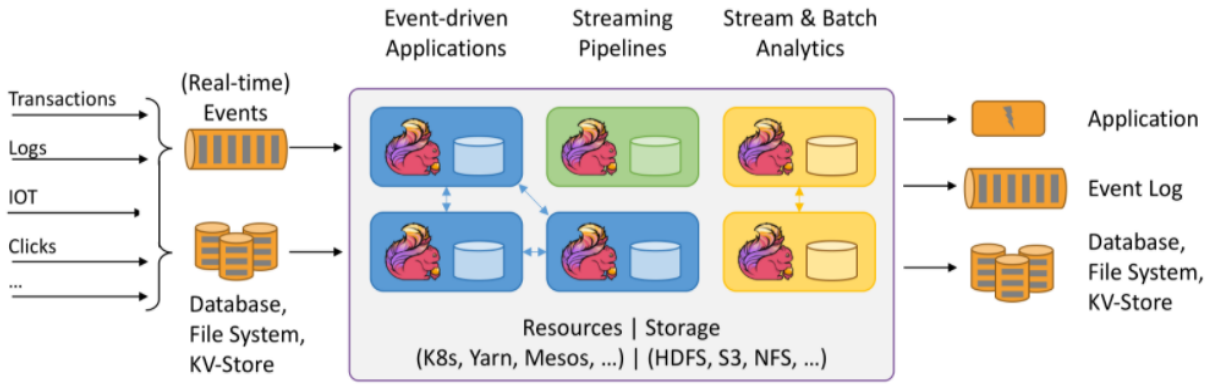
flink.apache.org

中国大学MOOC(慕... 首页 - 廖雪峰的官... 电子科技大学 GitHub 我的工作台 - 码云... (11 条消息) 首页 -... 哔哩哔哩 (゜- ゜)つ... JoTang

Flink

- What is Apache Flink?
- What is Stateful Functions?
- Use Cases
- Powered By
- Downloads
- Getting Started
- Documentation
- Getting Help
- Flink Blog
- flink-packages.org
- Community & Project Info

Apache Flink® — Stateful Computations over Data Streams



All streaming use cases

- Event-driven Applications
- Stream & Batch Analytics
- Data Pipelines & ETL

Learn more

Guaranteed correctness

- Exactly-once state consistency
- Event-time processing
- Sophisticated late data handling

Learn more

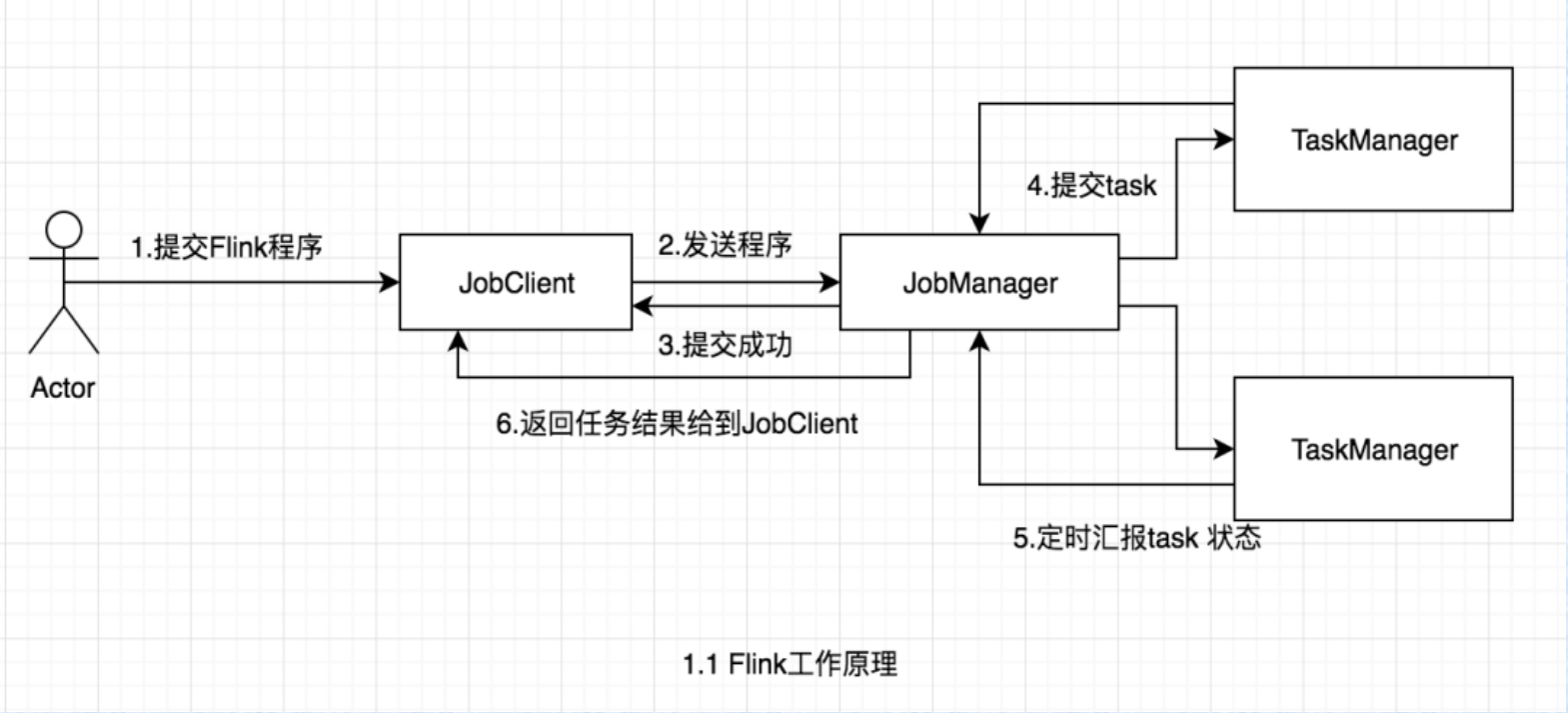
Layered APIs

- SQL on Stream & Batch Data
- DataStream API & DataSet API
- ProcessFunction (Time & State)

Learn more

Flink软件架构

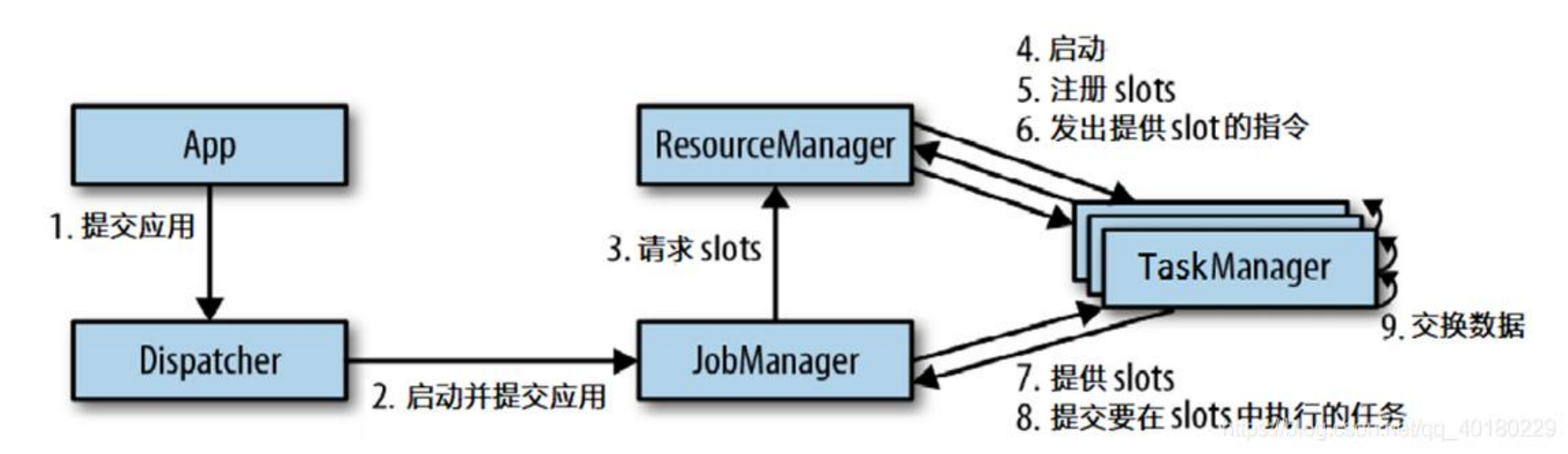
Flink整个系统主要由三个组件组成：JobClient, JobManager和TaskManager。Flink架构也遵循Master-Slave架构设计原则，JobManager为Master节点，TaskManager为Workwer（slave）节点。



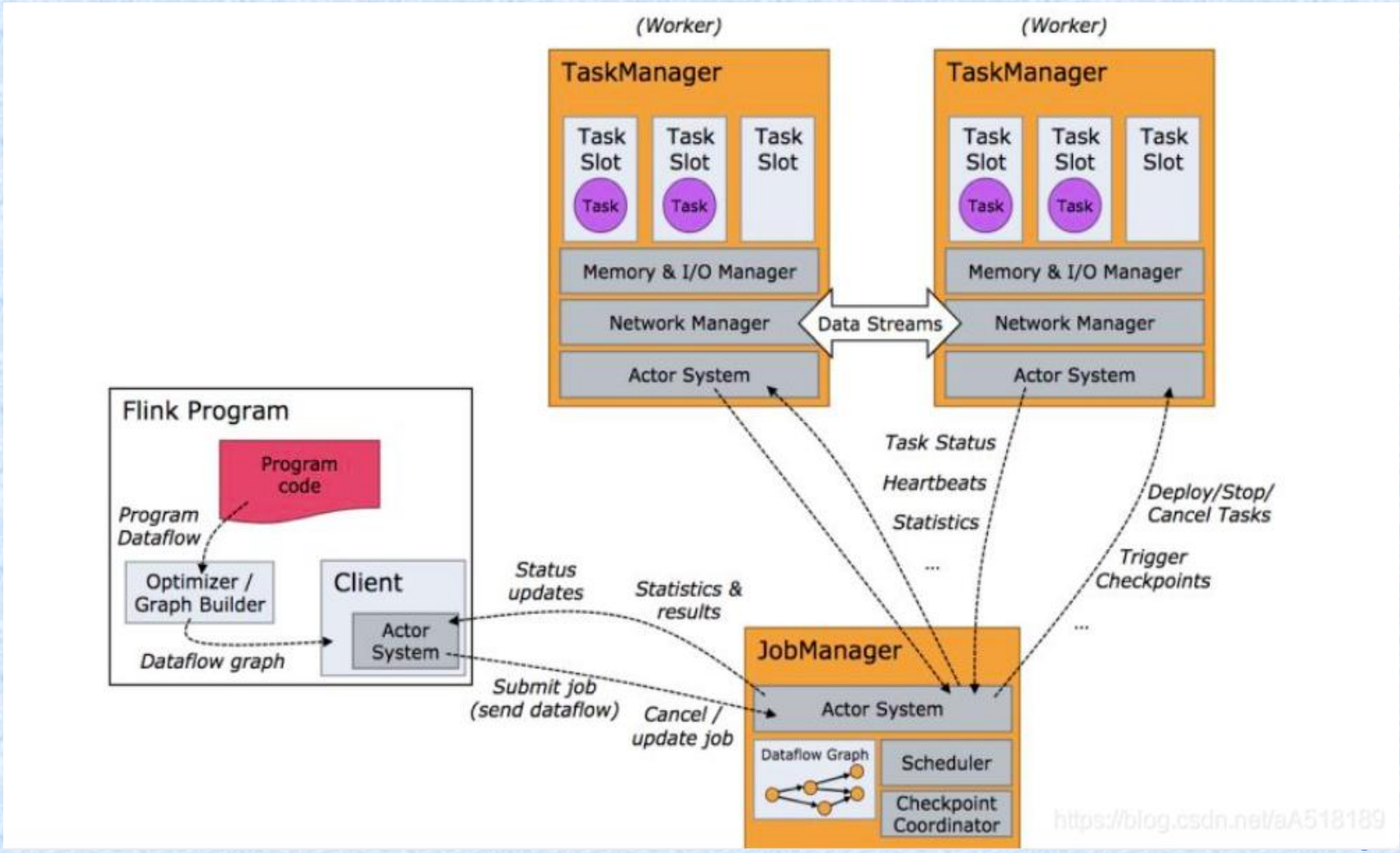
1.1 Flink工作原理

Flink软件架构（续）

另外还有Dispatcher组件 (不是必需的，取决于运行方式)，ResourceManager组件 (使用如YARN、Mesos、K8s等)



Flink工作原理


(Worker)
(Worker)
TaskManager
TaskManager
Task Slot
Task
Memory & I/O Manager
Network Manager
Actor System
Data Streams
Flink Program
Program code
Program Dataflow
Optimizer / Graph Builder
Client
Actor System
Dataflow graph
Status updates
Statistics & results
Submit job (send dataflow)
Cancel / update job
Task Status
Heartbeats
Statistics
...
Deploy/Stop/ Cancel Tasks
Trigger Checkpoints
...
JobManager
Actor System
Dataflow Graph
Scheduler
Checkpoint Coordinator

Flink工作原理

作业管理器（JobManager）

控制一个应用程序执行的主进程，也就是说，每个应用程序都会被一个不同的JobManager 所控制执行。

JobManager 会先接收到要执行的应用程序，这个应用程序会包括：作业图（JobGraph）、逻辑数据流图（logical dataflow graph）和打包了所有的类、库和其它资源的JAR包。

JobManager 会把JobGraph转换成一个物理层面的数据流图，这个图被叫做“执行图”（Execution Graph），包含了所有可以并发执行的任务。

JobManager 会向资源管理器（ResourceManager）请求执行任务必要的资源，也就是任务管理器（TaskManager）上的插槽（slot）。一旦它获取到了足够的资源，就会将执行图分发到真正运行它们的 TaskManager上。而在运行过程中，JobManager会负责所有需要中央协调的操作，比如说检查点（checkpoints）的协调。

任务管理器（TaskManager）

Flink中的工作进程：通常在Flink中会有多个TaskManager运行，每一个TaskManager都包含了一定数量的插槽（slots）。插槽的数量限制了TaskManager能够执行的任务数量。

启动之后，TaskManager会向资源管理器注册它的插槽；收到资源管理器的指令后，TaskManager就会将一个或者多个插槽提供给JobManager调用。JobManager就可以向插槽分配任务（tasks）来执行了。

在执行过程中，一个TaskManager可以跟其它运行同一应用程序的TaskManager交换数据。

资源管理器（ResourceManager）

主要负责管理TaskManager的插槽（slot），TaskManger 插槽是Flink中定义的处理资源单元。

Flink为不同的环境和资源管理工具提供了不同资源管理器，比如YARN、Mesos、K8s，以及standalone部署。

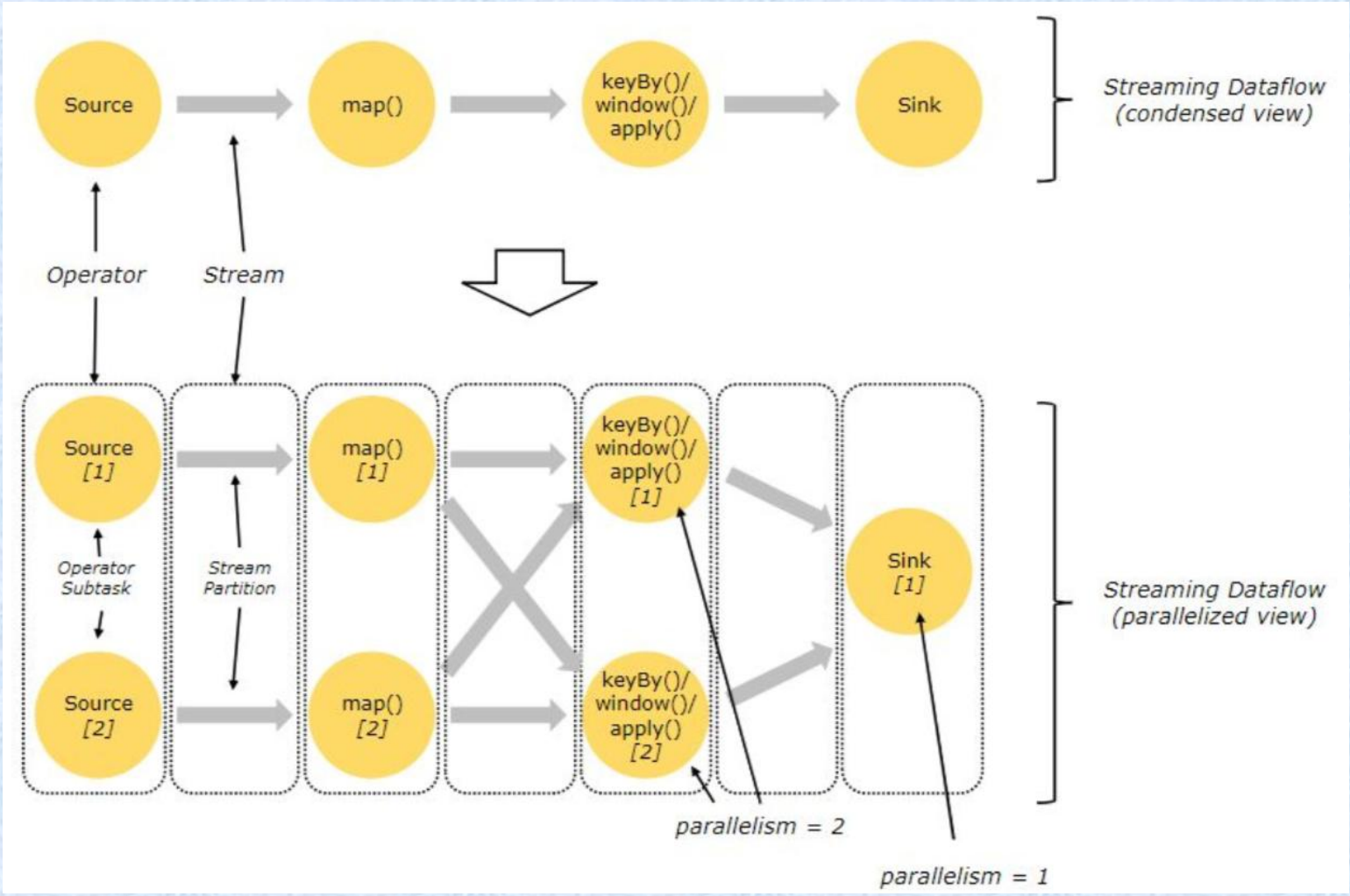
分发器（Dispatcher）

可以跨作业运行，它为应用提交提供了REST接口。

当一个应用被提交执行时，分发器就会启动并将应用移交给一个JobManager。

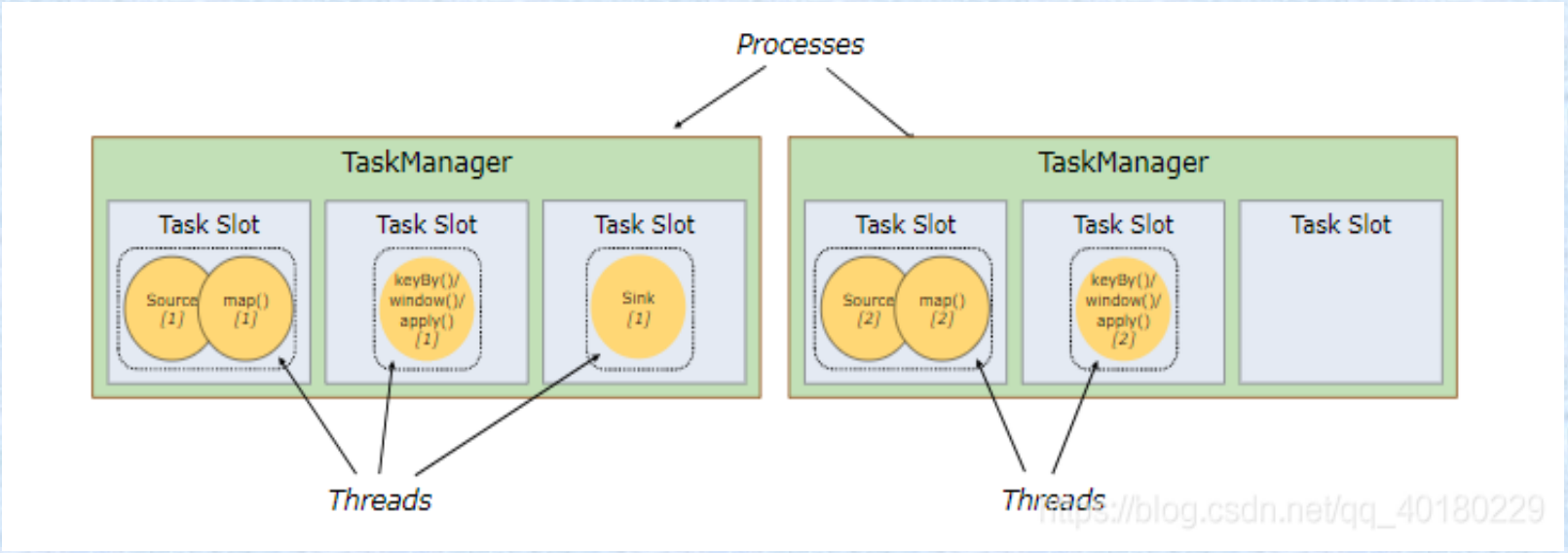
Dispatcher在架构中可能并不是必需的，这取决于应用提交运行的方式。

Flink并行模式

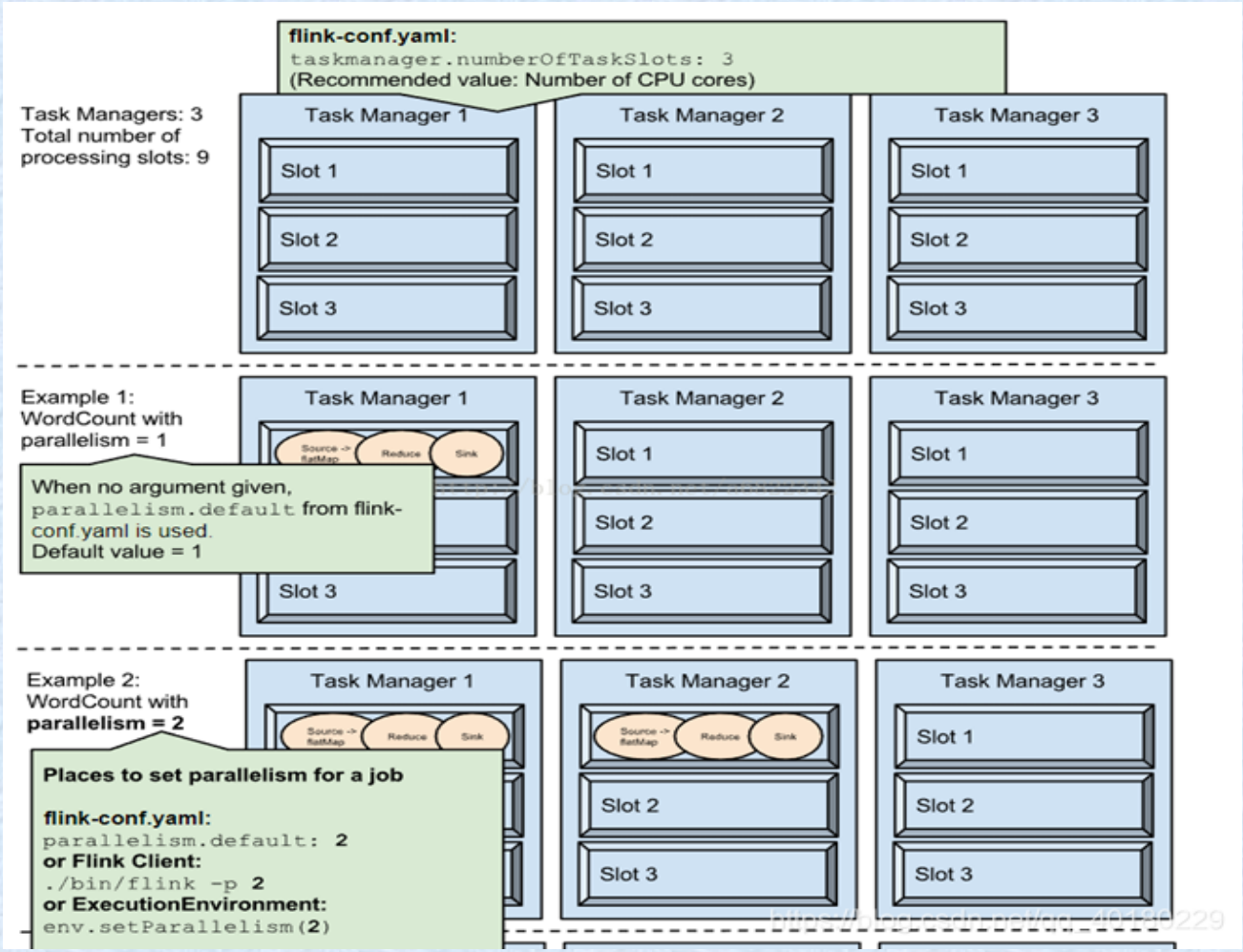

Source
map()
keyBy()/ window()/ apply()
Sink
Streaming Dataflow (condensed view)
Operator
Stream
Source [1]
Source [2]
map() [1]
map() [2]
keyBy()/ window()/ apply() [1]
keyBy()/ window()/ apply() [2]
Sink [1]
Operator Subtask
Stream Partition
Streaming Dataflow (parallelized view)
parallelism = 2
parallelism = 1

- Flink 中每一个 TaskManager 都是一个JVM进程，它可能会在独立的线程上执行一个或多个子任务
- 一个 TaskManager 能接收多少个 task， TaskManager 通过 task slot 来进行控制（一个 TaskManager 至少有一个 slot）
- 图中每个Task Manager中的Slot为3个，那么两个Task Manager一共有六个Slot, 而这6个Slot代表着Task Manager最大的并发执行能力，一共能可以6个task并行执行
- Slot是静态概念，代表着Task Manager具有的并发执行能力，可以通过参数taskmanager.numberOfTaskSlots进行配置
- 图中Source和Map是一个Task，且并行度(我们设置的setParallelism())都为1，指这个task任务的并行能力为1，只占用一个Slot资源

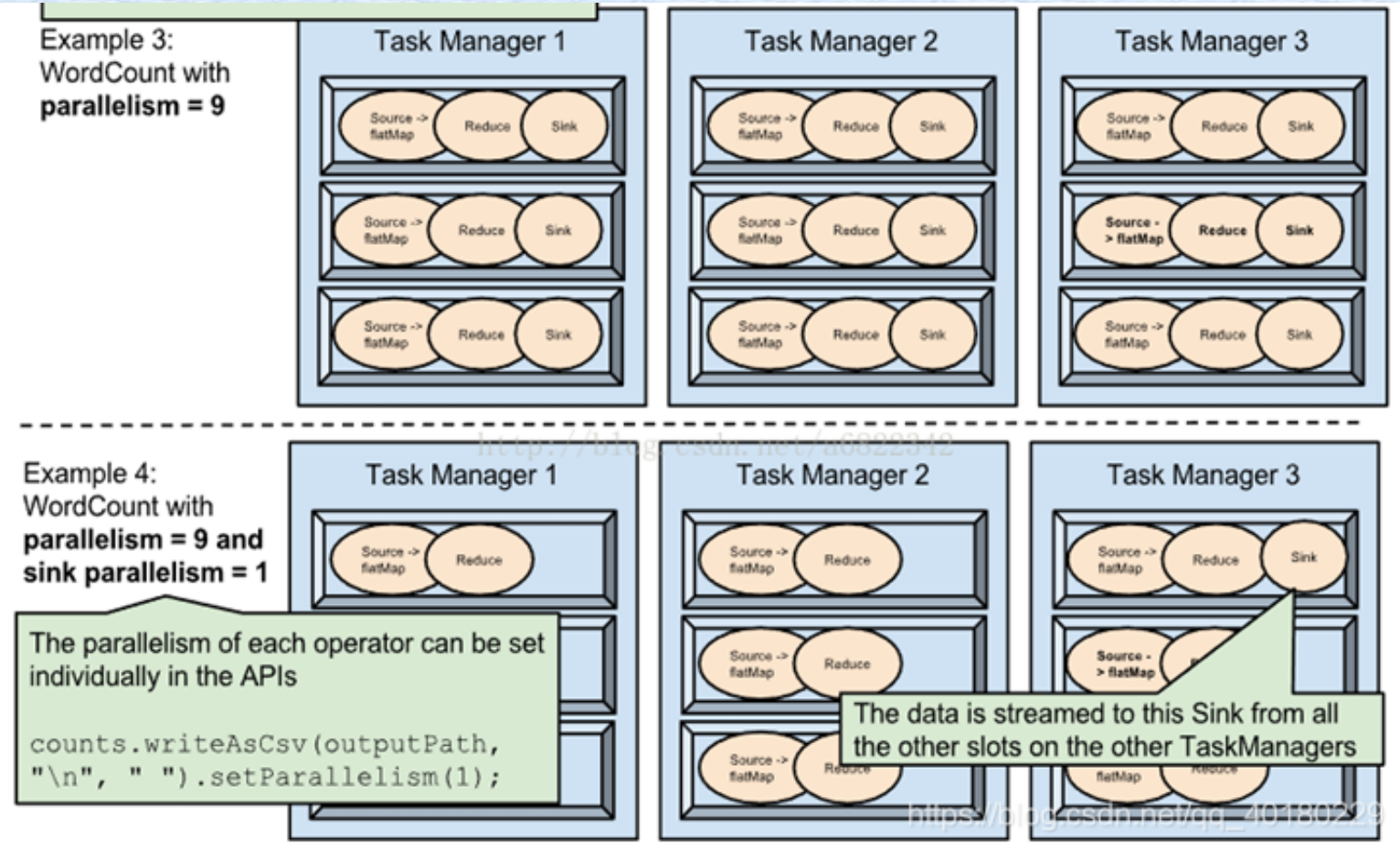
- 如果TaskManager只有一个slot，那将意味着每个task group运行在独立的JVM中（该JVM可能是通过一个特定的容器启动），而TaskManager有多个slot意味着更多的subtask可以共享同一个JVM。而在同一个JVM进程中的task将共享TCP连接（基于多路复用）和心跳消息。它们也可能共享数据集和数据结构，因此这减少了每个task的负载。
- TaskManager的并行度可以通过参数parallelism.default进行配置。



- 假设一共有3个TaskManager，每一个TaskManager中的分配3个TaskSlot，也就是每个TaskManager可以接收3个task，一共9个TaskSlot



- 一个特定算子的 子任务（subtask）的个数被称之为其并行度parallelism，我们可以单独对每个算子设置并行度，也可以直接用env设置全局并行度。



大数据计算技术
Big Data Computing Technology

	Storm	Spark Streaming	Samza	Flink
流处理模式	Native Stream Processing	Micro-batch Stream Processing	Native Stream Processing	Native Stream Micro-batch Structured data
数据模型	Tuple（多元组）	Dstream （RDD组成的离散流）	单条消息	Dataflow
数据源	Spout	Spark Streaming	Kafka Consumer	Data Streams （多种Sources）
处理单元	Bolt	Task	Task	Task
并行模式	基于Topology的多节点多任务并行模式	基于RDD多节点多任务并行模式	基于分区队列的多节点多任务并行模式	数据流分块，任务拆解为子任务。操作子任务是相互独立的，操作子任务的数量决定并行度
状态维护	Stateless 需要自己写或使用Trident	Stateful Spark Streaming提供状态维护API	Stateful 通过本地存储和Kafka Changelog来实现	Stateful 存储在State Backend中优先内存，其次硬盘
响应延迟	毫秒级	秒级（取决于batch设置 大小）	毫秒级	可选（Processing-time Mode）
编程语言	Java, Python, Ruby, JavaScript, Perl	Java, Python, Scala	Java, Scala	Java, Scala, SQL, python