13

流计算

13.1 流计算模型

流计算模型

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，$\alpha i\ (s) = 0$。

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

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

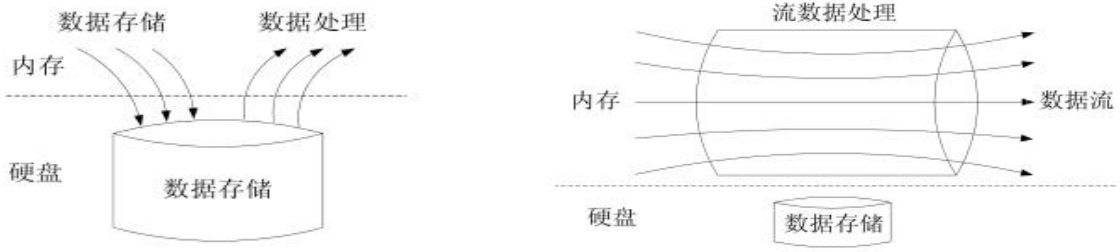
针对上述流数据类型的计算模式称为流计算（Stream Computing）。

13.1 流计算模型

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

13.1 流计算模型

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。

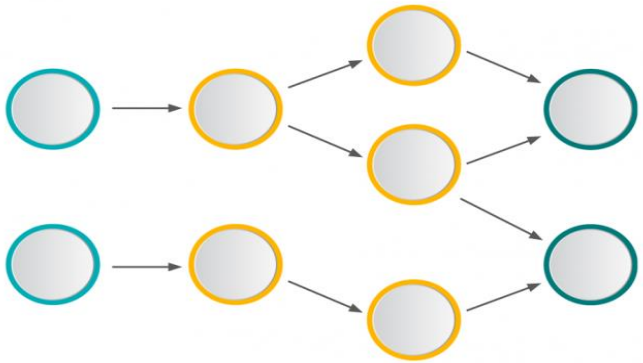


图 15-2 有向非循环向图（DAG）

13.1 流计算模型

流计算两种主要处理模式

- 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。

13.1 流计算模型

Native Stream Processing System是基于数据按其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），系统响应性好。

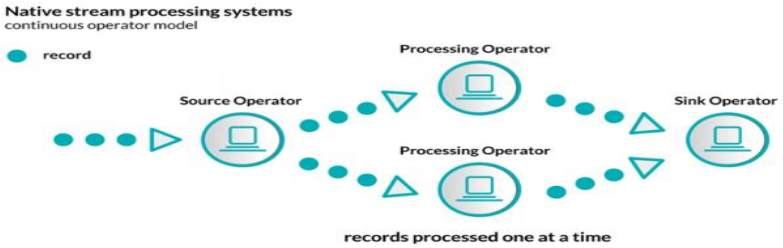


图 15-3　Native Stream Processing System

Micro-batch Stream Processing System是将数据流先作预处理，打包成含多条数据的batch（批次）再交给系统处理。逐条处理模式简便易行，系统延迟性也是最低的，但它至少存在两个问题：一是系统吞吐率（throughput）低；二是容错成本高和容易负载不均衡。

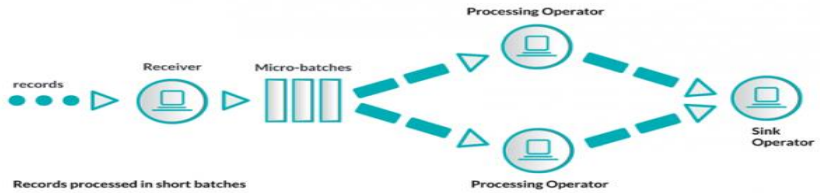


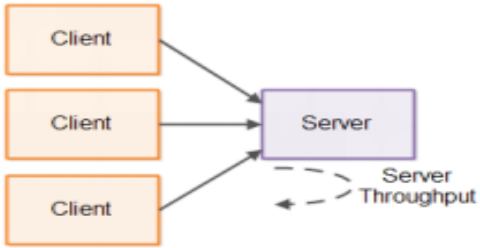
图 15-4　Micro-batch Stream Processing System

13.1 流计算模型

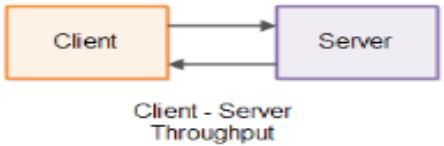
流计算性能参数

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。



（a）服务器端吞吐率



（b）客户端吞吐率

流计算性能参数

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。

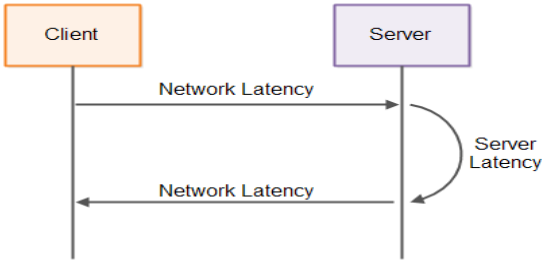


图 15-6　Client/Server 系统响应时间

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。

13.1 流计算模型

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：

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

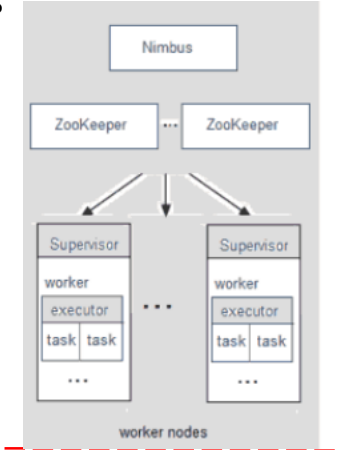
$1/(2L_n + L_s) < 1/(0.2L_n + L_s)$

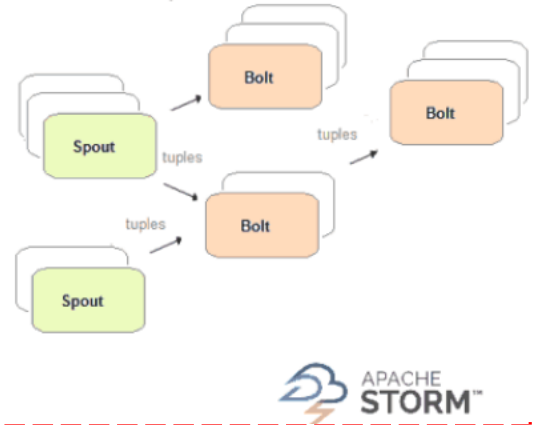
由此可知，两种模式比较，Native Stream Processing System的时间延迟性好于Micro-batch Stream Processing System，但系统吞吐率则低于Micro-batch Stream Processing System。另外，Native Stream Processing System容错性成本也较高，因为逐条数据备份或恢复的开销大于成批次的处理成本。

流并行计算实现

Storm的Topoloy模型

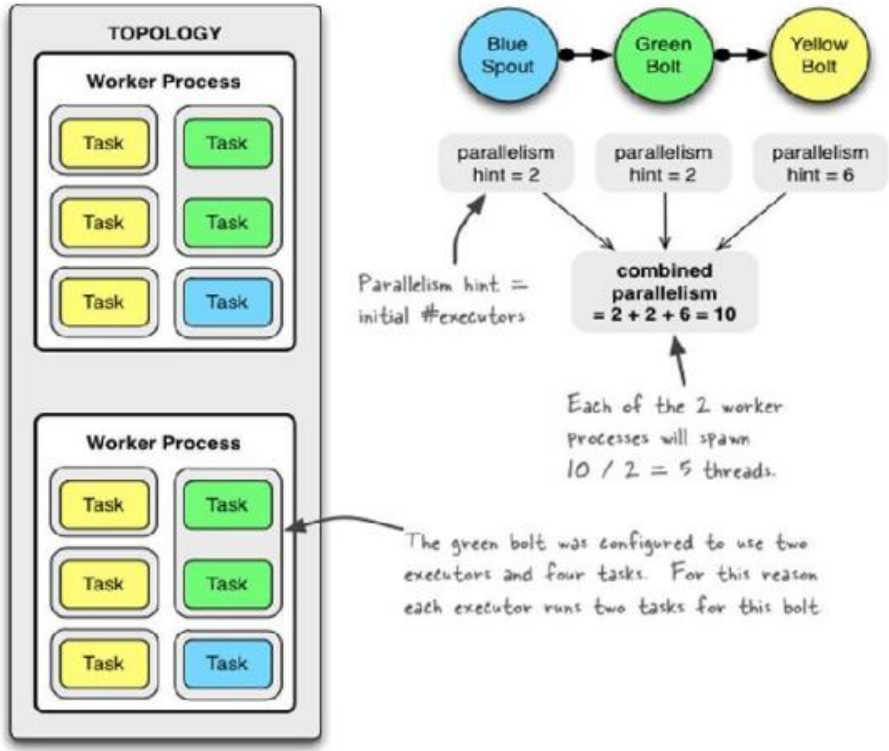
Twitter的Storm仍然是一种Native Stream Processing System，即对流数据的处理是基于每条数据进行，其并行计算是基于由Spout（数据源）和Bolt（处理节点）组成的有向拓扑图Topology来实现，如图15-9所示。这里，流数据是以Tuple（基本数据单元，可看作一组各种类型的值域组成的多元组）的形式在Spout与Bolt之间流转。Spout负责将输入数据流转换成一个个Tuples,发送给Bolt处理。每个Bolt读取上游传来的Tuples，向下游发送处理后的Tuples。





13.1 流计算模型

- 当一个Storm作业被提交时，同时需要提交预先设计的Topology（包含了Spout和Bolt的信息）。由于Topology里的Spout和Bolt的功能最终是靠Worker节点上的Task来实现的，而且一个Spout或Bolt的任务需要分布在不同Worker上的多个Task来并行完成，这就还需要确定每个Spout和Bolt需要多少个Task来支撑，以及如何把一个Spout或Bolt映射到多个Worker节点的Task上去，如图15-10所示。



13.1 流计算模型

Spark的DStream模型

- Spark 流计算的核心概念是 Discretized Stream（DStream）。如图15-11所示，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 Streaming 的计算分析可以基于单个RDD，也可以基于DStream上的滑动window，即通过移动一个固定长度的window来读取DStream的某一段RDDs。

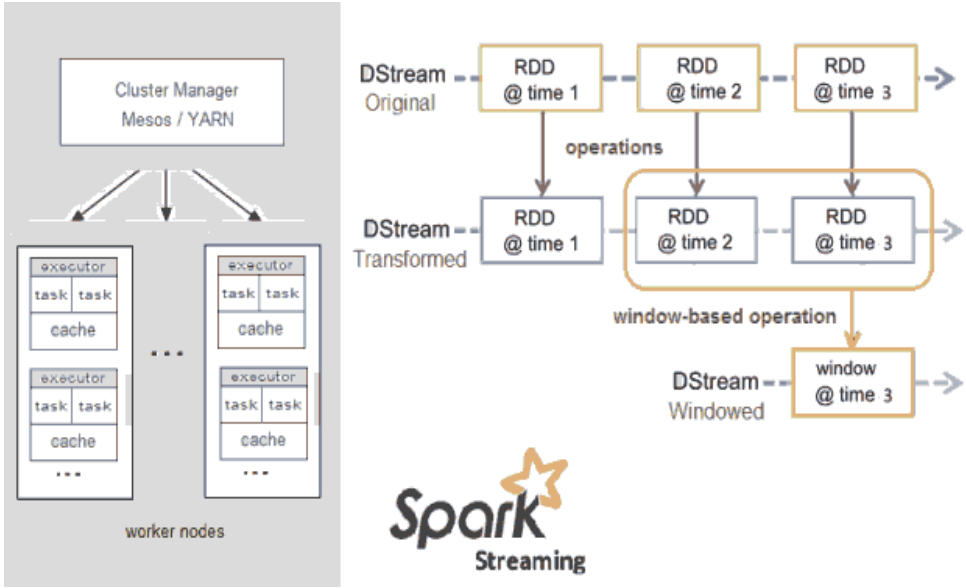


图 15-11　Spark 的并行计算模型

13.1 流计算模型

●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，如图15-12所示。Driver与Hadoop集群的管理程序如Mesos[12]，YARN[13]进行对接，负责把应用程序的计算任务转化成有向非循环图（DAG），而Executor则负责完成Worker节点上的计算和数据存储。Spark Streaming的并行处理是基于DStream设计的。

各个Worker节点上的Executor计算是基于DStream进行，DStream所包含的RDD在进行分区（partition）后分发给各个Executor，针对一个个数据partition再由Executor生成一个个Task线程，这些Task线程在Worker节点上并行运行完成计算任务。

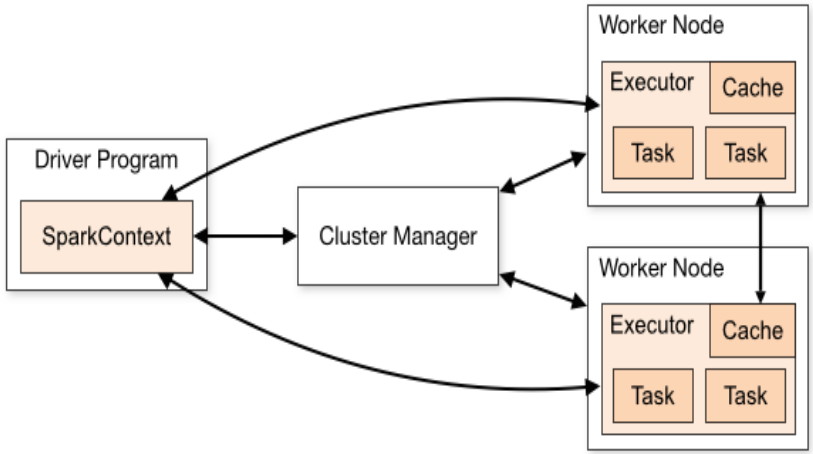
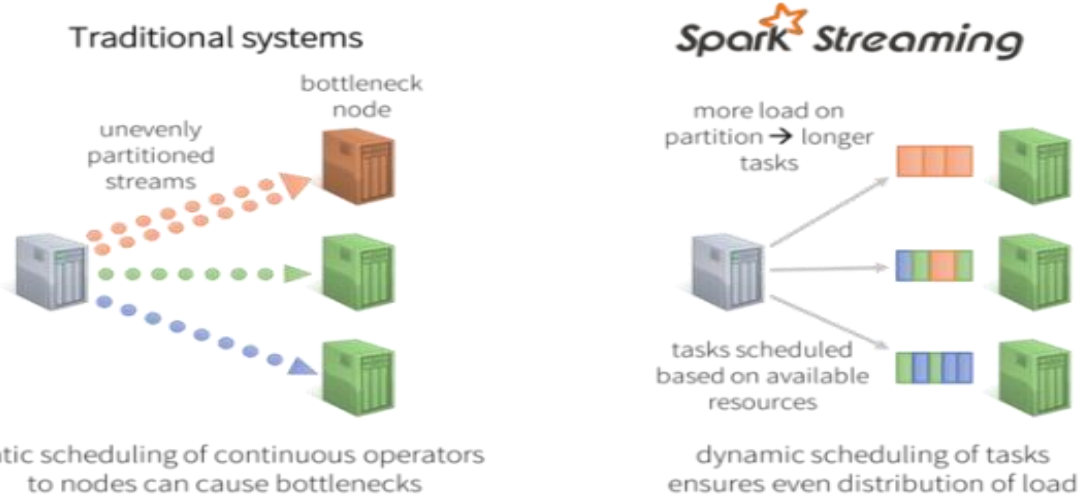


图 15-12　Spark 计算体系

13.1 流计算模型

●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 node（图15-13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15-13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。



13.2 Storm计算架构

●作为一个流计算框架，Storm具备如下特点：
分布式：具有水平扩展能力（通过增加集群机器和并发数提升计算能力）
实时性：对流数据的快速响应处理，响应时延可控制在毫秒级
数据规模：支持海量数据处理，数据规模可达TB甚至PB量级
容错性：提供系统级的容错和故障恢复机制
简便性：简单的编程模型，支持编程语言如Java，Clojure，Ruby，Python，要增加对其他语言的支持，只需实现一个简单的Storm通信协议即可

13.2 Storm计算架构

●Storm逻辑架构

● Storm的计算架构分为逻辑架构（抽象模型）与物理架构（系统结构）两个方面。逻辑架构主要包含以下组件：

数据模型 Tuple
数据流 Stream
数据源 Spout
处理单元 Bolt
分发策略 Stream Grouping
逻辑视图 Topology

13.2 Storm计算架构

●多元组Tuple

● Tuple是由一组各种类型的值域组成的多元组，所有的基本类型、字符串以及字节数组都作为Tuple的值域类型，也可以使用用户自己定义的类型，它是Storm的基本数据单元，如图15-18所示。

图15-18 Tuple格式

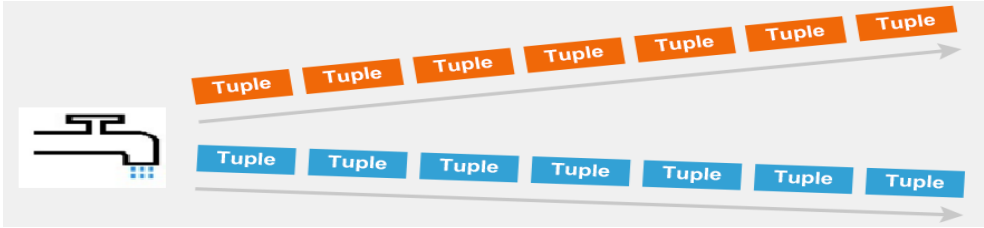
●数据流Stream

● Stream是一个不间断的无界的连续Tuple序列，是Storm对流数据的抽象，如图15-19所示。

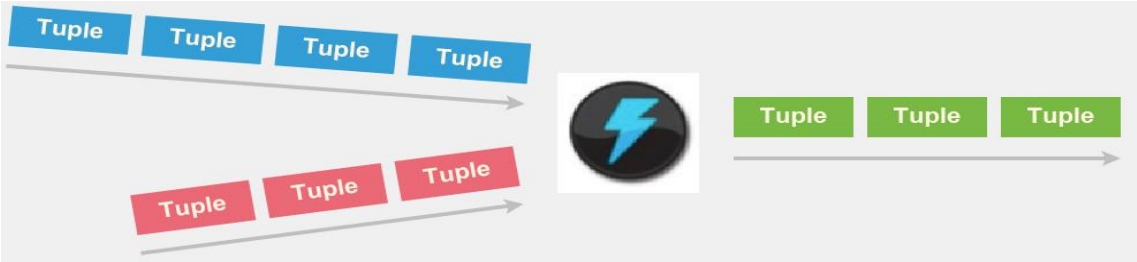
图15-19 Stream组成

13.2 Storm计算架构

- 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。



13.2 Storm计算架构

消息分发策略Stream Grouping

● Tuple序列从上游Bolt到某个下游Bolt其多个并发Task的分组分发方式，如图15-22所示。

Shuffle Grouping: 随机分组。
Fields Grouping：按字段分组。
All Grouping：广播发送。
Global Grouping: 全局分组。
Non-Grouping: 不分组。
Direct Grouping: 直接分组。

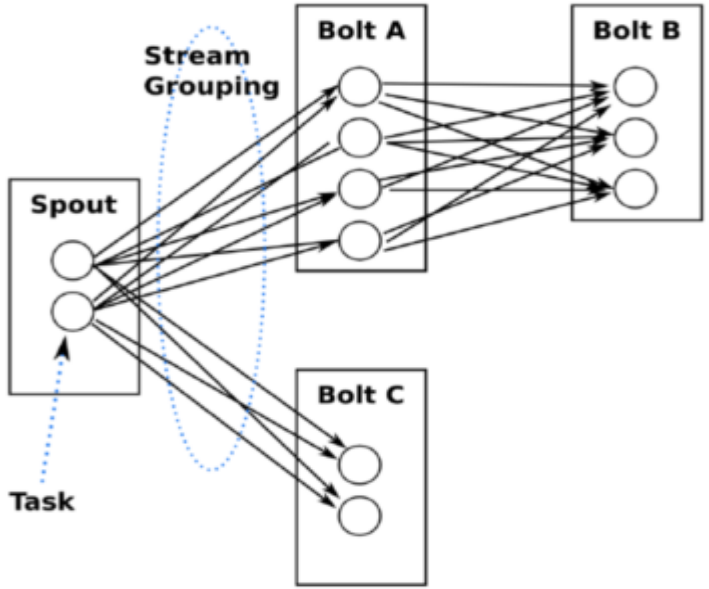


图15-22 分发策略Stream Grouping

13.2 Storm计算架构

逻辑视图Topology

Topology是一个由Spout源，Bolt节点，Tuple流，Stream Grouping分发方式组成的一个有向图（DAG），代表了一个Storm作业（Job）的逻辑架构，如图15-23所示。

Storm对数据的处理逻辑与算法封装在Bolt里，那么一个Storm作业的计算流程就封装在Topology里。因此，一个设计好的Topology可以提交到Storm集群去执行。

Topology只是一个Storm作业流程的逻辑设计，真正要实现这个逻辑设计，还需要Storm的系统架构或物理模型来支撑。

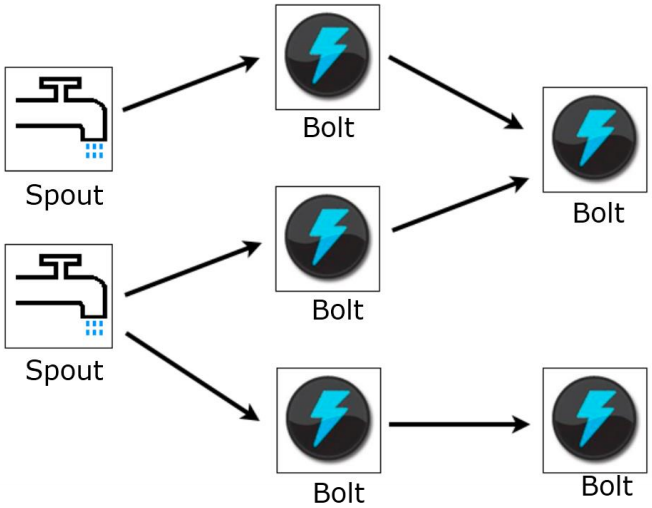
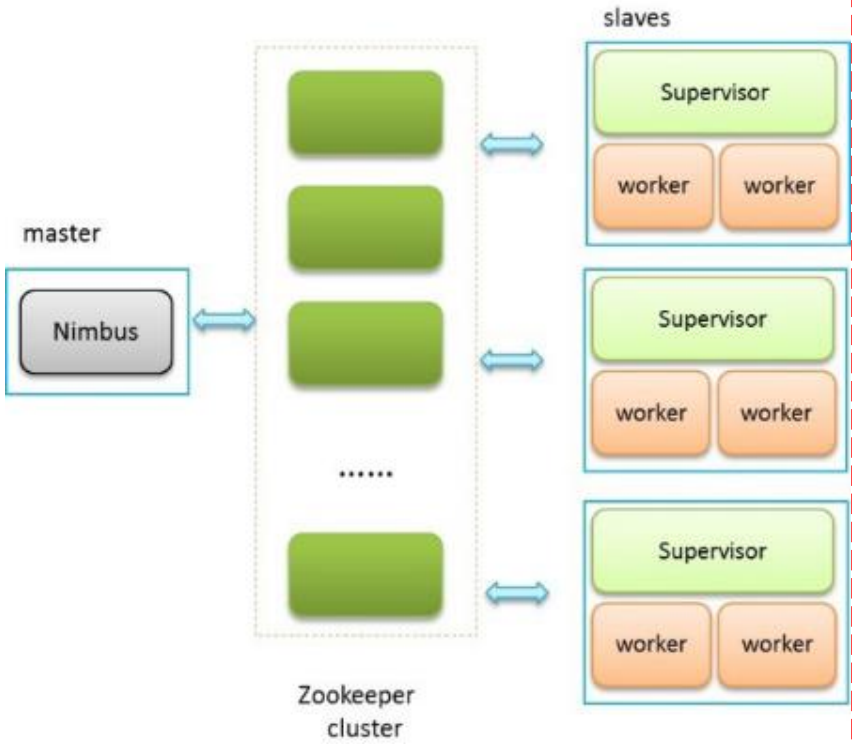


图 15-23　Topology 视图

13.2 Storm计算架构

Storm的计算体系也采用了主从（Master/Slave）架构，主要有两类节点：主节点Master和工作节点Slaves，如图15-24所示。主节点上运行一个叫做Nimbus的守护进程，类似于Hadoop的JobTracker，负责集群的任务分发和故障监测。Nimbus通过一组Zookeeper管理众多的工作节点。每个工作节点运行一个叫做Supervisor的守护进程，监听本地节点状态，根据Nimbus的指令在必要时启动和关闭本节点的工作进程。



13.2 Storm计算架构

Storm的系统架构（物理视图）包含如下组件：

Storm主控程序	Nimbus
集群调度器	Zookeeper
工作节点控制程序	Supervisor
工作进程	Worker
执行进程	Executor
计算任务	Task

主控程序Nimbus

运行在主节点上，是整个流计算集群的控制核心，总体负责topology的提交、运行状态监控、负载均衡及任务重新分配等。Nimbus分配的任务包含了Topology代码所在路径(在Nimbus本地节点上)以及Worker， Executor和Task的信息。

集群调度器Zookeeper

由Hadoop平台提供，是整个集群状态同步协调的核心组件。Supervisor，Worker，Executor等组件会定期向Zookeeper写心跳信息。当Topology出现错误或者有新的Topology提交到集群时，相关信息会同步到Zookeeper。

13.2 Storm计算架构

工作节点控制程序 Supervisor

运行在工作节点（称为node）上的控制程序，监听本地机器的状态，接受Nimbus指令管理本地的Worker进程。Nimbus和Supervisor都具有fail-fast（并发线程快速报错）和无状态的特点。

工作进程 Worker

运行在node上的工作进程。Worker由node + port唯一确定，一个node上可以有多个Worker进程运行，一个Worker内部可执行多个Task。Worker还负责与远程node的通信。

执行进程 Executor

提供Task运行时的容器，执行Task的处理逻辑。一个或多个Executor实例可以运行在一个Worker中，一个或多个Task线程也可运行在一个Executor中，如图15-26所示。

计算任务 Task

逻辑组件Spout/Bolt在运行时的实体，也是Executor内并行运行的计算任务。一个Spout/Bolt在运行时可能对应一个或多个Tasks，并行运行在不同节点上。Task数目可在Topology中配置，一旦设定不能改变。

13.2 Storm计算架构

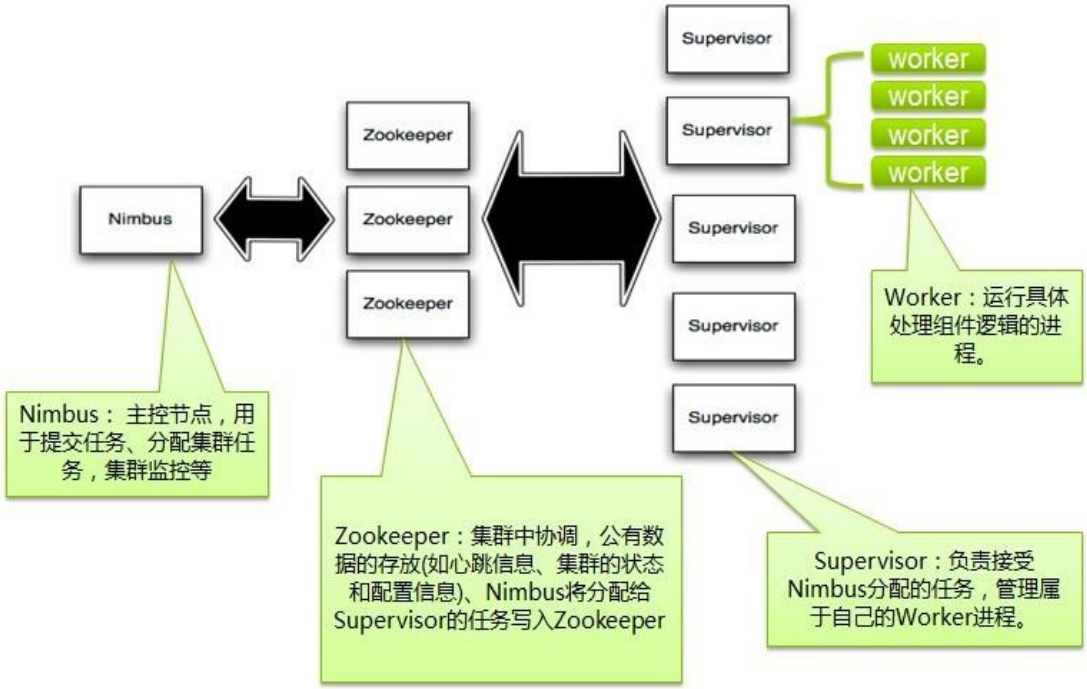


图 15-25　Storm 的技术架构

13.3 Storm工作机制

Topology提交与执行

Storm作业Topology的提交过程如图15-28所示。在非本地模式下，客户端通过Thrift调用Nimbus接口来上传代码到Nimbus并启动提交操作。Nimbus进行任务分配，并将信息同步到Zookeeper。Supervisor定期获取任务分配信息，如果Topology代码缺失，会从Nimbus下载代码，并根据任务分配信息同步Worker。Worker根据分配的tasks信息，启动多个Executor线程，同时实例化Spout，Bolt，Acker等组件，待所有connections（Worker和其它机器通讯的网络连接）启动完毕，此Storm系统即进入工作状态。Storm的运行有两种模式: 本地模式和分布式模式。

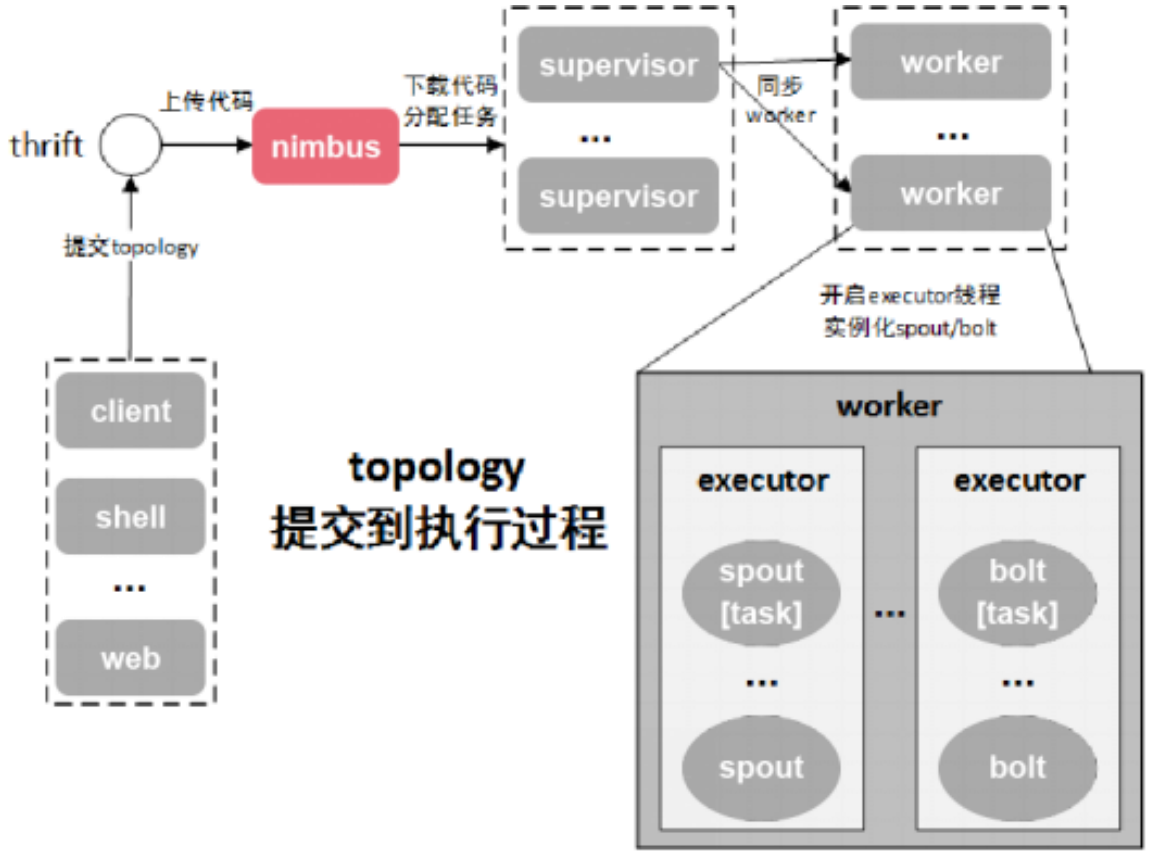
1) 本地模式

Storm用一个进程里面的线程来模拟所有的Spout和Bolt。本地模式只对开发测试来说有用。

2) 分布式模式

Storm以多进程多线程模式运行在一个集群上。当提交Topology给Nimbus的时候， 同时就提交了Topology的代码。Nimbus负责分发你的代码并且负责给你的topolgoy分配工作进程，如果一个工作进程挂掉，Nimbus会把它重新分配到其它节点。

13.3 Storm工作机制



消息发送ACK机制

Storm可靠性要求发出的Spout每一个tuple都会完成处理过程，其含义是这个tuple以及由这个tuple所产生的所有后续的子tuples都被成功处理。由于Storm是一个实时处理系统，任何一个消息tuple和其子tuples如果没有在设定的timeout时限内完成处理，那这个消息就失败了，因此Storm需要一种ACK（Acknowledgement）机制来保证每个tuple在规定时限内得到即时处理。这个timeout时限可以通过Config.TOPOLOGY_MESSAGE_TIMEOUT_SECS来设定，Timeout的默认时长为30秒。

Tuple Tree的构成

以图15-29的Tuple Tree为例，输入tuple A在Bolt处完成了处理，并向下游发送了2个衍生tuples B和C，在Bolt向跟踪的Acker报告了Ack后，Tuple Tree就只包含了tuples B和C（tuple A打红X表示它已不在当前状态的Tuple Tree中）。

然后tuple C流转到下一个Bolt，被处理完后又衍生了tuples D和E。该Bolt向Acker确认已处理完tuple C，于是C被移出Tuple Tree，当前状态的Tuple Tree变成只包含B，D，E 。。。这一过程将持续进行，直到没有新的tuple加入这个Tuple Tree，而树中所有的tuples都完成了处理移出了Tuple Tree。

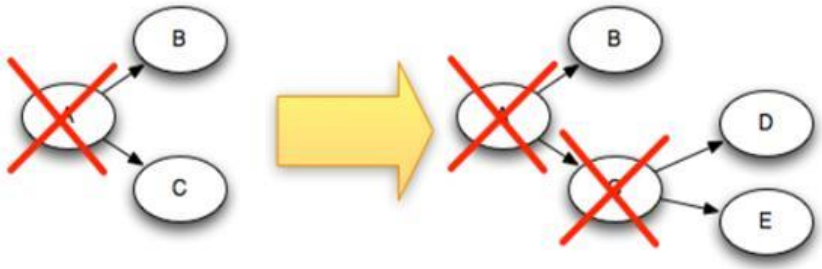


图 15-29　Tuple Tree 的更新

Acker算法

前面提到，一个Spout发出的tuple的Tuple Tree构成和更新是由处理该tuple的各个Bolts在流转过程中完成，跟踪这个tuple及其衍生tuples（它们构成了Tuple Tree）的Acker程序最终基于以下算法判断Tuple Tree是否处理完毕（即树中所有的节点都被Acked），也即判断该tuple处理是否结束：

1）当Spout生成一个新tuple时，会向Acker发送如下一条信息通知Acker跟踪：

{ spout-tuple-id {:spout-task task-id : val ack-val } }

这里，spout-tuple-id：这条新tuple随机生成的64-bit ID

task-id：产生这条tuple的Spout ID，Spout可能有多个task，每个task都会被分配一个唯一的taskId

ack-val：Acker使用的64-bit的校验值，初始值为0

收到Spout发来的初始tuple消息后，Acker首先将ack-val（此时为0）与初始tuple的msgId做一个XOR（exclusive OR）运算（表15.5），并将结果更新ack-val值：

ack-val = (ack-val) XOR (spout-tuple-id);

13.3 Storm工作机制

表 15.5 二进制的 XOR 运算符定义

Operand	运算符	Operand	结果值
0	XOR	0	0
0		1	1
1		0	1
1		1	0

2）Bolt处理完输入的tuple，若创建了新的衍生tuples向下游发送，在向Acker发送消息确认输入tuple完成时，它会先把输入tuple的msgId与所有衍生tuples的msgId（也是64-bit的全新ID）作XOR运算，然后把结果tmp-ack-val包含在发送的Ack消息中，消息格式是

:(spout-tuple-id, tmp-ack-val)

Acker收到每个Bolt发来的Ack消息，都会执行如下运算：

ack-val = (ack-val) XOR (tmp-ack-val);

所以ack-val所含值总是目前Tuple Tree中所有tuples的msgId的XOR运算值。

3）当Acker收到一个Ack消息使ack-val = 0时，该条tuple的处理结束，因为(ack-val) XOR (tmp-ack-val) = 0意味着ack-val的值与tmp-ack-val相同（只有两个值完全相同时XOR的运算结果才为0）。这就意味着整个Tuple Tree在规定时间内（timeout）再无新的tuple产生，整个运算结束。

有无可能由于两个衍生tuple的ID值碰巧相同，造成ack-val在Tuple Tree处理完之前就变成0？由于衍生tuple也是64-bit的随机数，两个64-bit随机生成的ID值完全一样的概率非常低，几乎可忽略不计，因此在Tuple Tree处理完之前ack-val为0的概率非常小；

4）根据最后的tuple处理成功或失败结果，Acker会调用对应的Spout的ack()或fail ()方法通知Spout结果，如果用户重写了ack ()和fail ()方法，Storm就会按用户的逻辑来进行处理。

13.3 Storm工作机制

下面我们以图15-30的Topology Tree为例讲解Acker算法流程。该Topology包含1个Spout，3个Bolts，流程步骤如下：
步骤一：Spout读入数据后生成了2个tuples（msgId分别为1001和1010），通知Acker；
步骤二：tuple 1001流入Bolt1，处理完后产生了新的tuple 1110，Bolt1向Acker发送了tuple 1001的Ack；
tuple 1010流入Bolt2，处理完后产生了新的tuple 1111，Bolt2向Acker发送了tuple 1010的Ack；
步骤三：两个tuples 1110，1111流向Bolt3，处理完后不再有新tuple产生，Bolt3向Acker发送了处理结果的Ack。

13.3 Storm工作机制

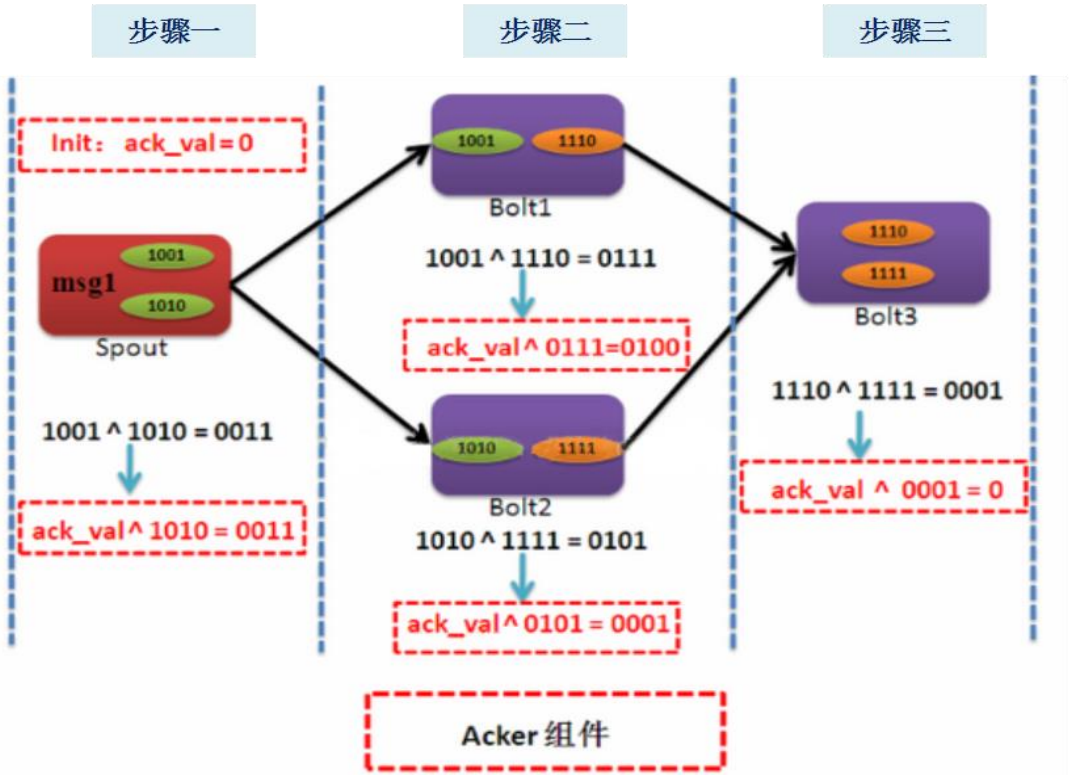


图 15-30　Acker 算法算例

ACK关闭

在某些场景下我们不希望使用ACK可靠性机制，或者对一部分流数据不需要保证处理成功，可以用如下方式关闭或部分关闭ACK功能：

1.把Config.TOPOLOGY_ACKERS设置成0。在这种情况下，Storm会在Spout发射一个tuple之后马上调用Spout的ack ()方法，这样这个Tuple整个的Tuple Tree不会被跟踪;

2.也可在Spout发射tuple的时候不设定msgId来达到不跟踪这个tuple的目的，这种发射方式是一种不可靠的发射；

3.如果对于一个Tuple Tree的某一部分tuples是否处理成功不关注，可以在Bolt发射这些Tuple的时候不锚定它们。这样这部分tuples就不会加入到Tuple Tree里面，也就不会被跟踪了。

容错机制

Storm从任务（线程）、组件（进程）、节点（系统）三个层面设计了系统容错机制，尽可能实现一种可靠的服务。

1. 任务级容错（Task）

如果Bolt Task线程崩溃，导致流转到该Bolt的tuple未被应答。此时Acker会将所有与此Bolt Task关联的tuples都设置为为超时失败，并调用对应的Spout的fail ()方法进行后续处理。

如果Acker Task本身失效，Storm会判定它在失败之前维护的所有tuples都因超时而失败，对应Spout的fail ()方法将被调用。

如果Spout任务失败，在这种情况下，与Spout对接的外部设备（如MQ队列）负责消息的完整性。例如当客户端异常时，外部kestrel队列会将处于pending状态的所有消息重新放回队列中。另外，Storm记录有Spout成功处理的进度，当Spout任务重启时，会继续从以前的成功点开始。

13.3 Storm工作机制

2. Slot故障（Process）

如果一个Worker进程失败，每个Worker包含的数个Bolt (或Spout) Tasks也失效了。负责监控此Worker的Supervisor会尝试在本机重启它，如果在启动多次仍然失败，它将无法发送心跳信息到Nimbus，Nimbus将判定此Worker失效，将在另一台机器上重新分配Worker并启动。

如果Supervisor失败，由于Supervisor是无状态的（所有的状态都保存在Zookeeper或者磁盘上）和fail-fast（每当遇到任何意外的情况，进程自动毁灭），因此Supervisor的失败不会影响当前正在运行的任务，只要及时将Supervisor重新启动即可。

如果Nimbus失败，由于Nimbus也是无状态和fail-fast的，因此Nimbus的失败不会影响当前正在运行的任务，只是无法提交新的Topology，只需及时将它重启即可。

3. 集群节点故障（Node）

如果Storm集群节点发生故障。此时Nimbus会将此节点上所有正在运行的任务转移到其他可用的节点上运行。

若是Zookeeper集群节点故障，Zookeeper自身有容错机制，可以保证少于半数的机器宕机系统仍可正常运行。