Lecture 17 Storm计算架构

- 逻辑架构
- 系统架构
- Storm实现机制

Storm应用场景

包括实时分析、在线机器学习、连续计算等

- 推荐系统：实时推荐，根据下单或加入购物车推荐相关商品
- 金融系统：实时分析股票信息数据
- 预警系统：根据实时采集数据，判断是否到了预警阈值
- 网站统计：实时销量、流量统计，如淘宝双11效果图

携程-网站性能监控

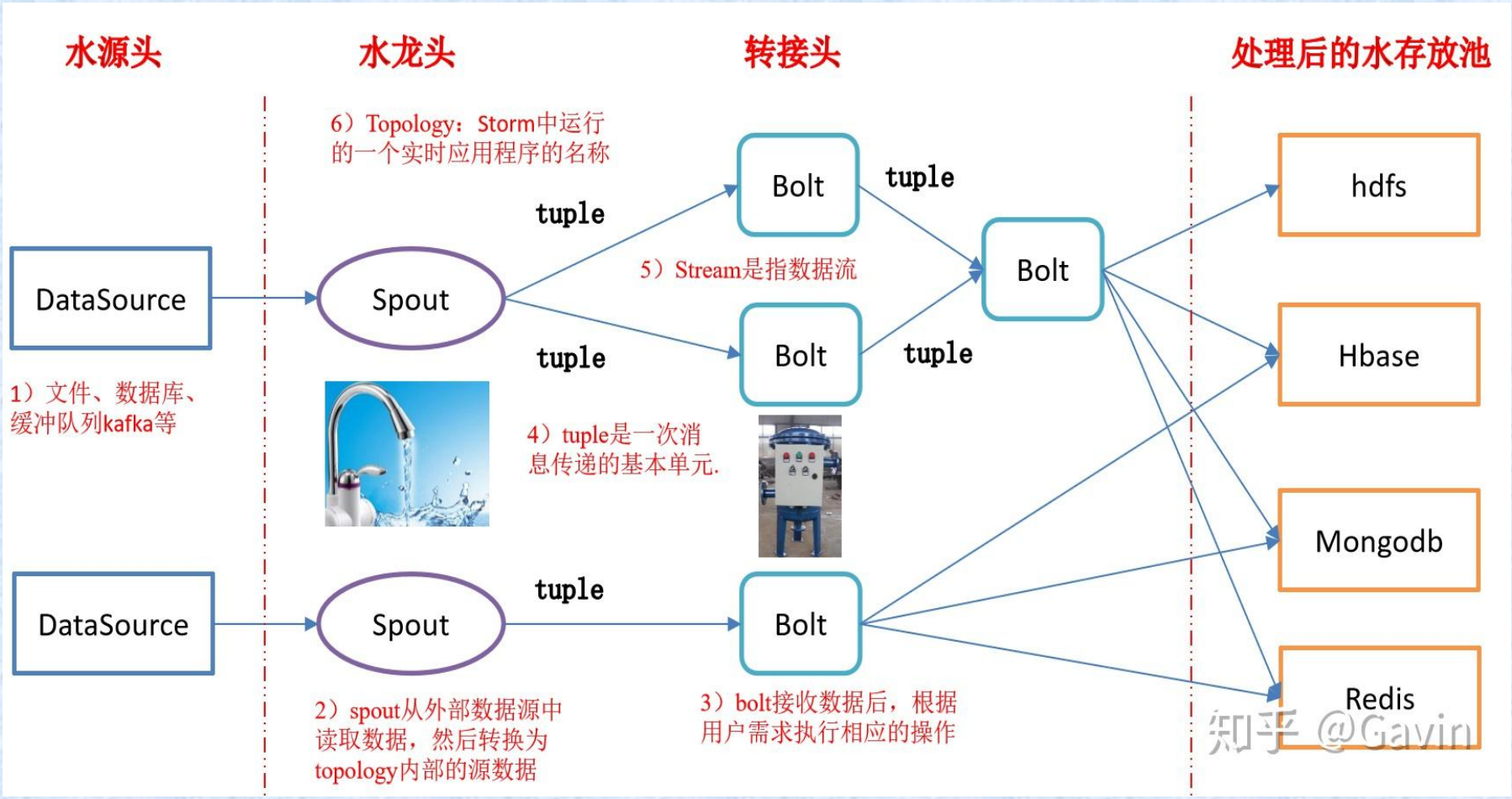

总览
监控中心
用户中心
xiaofei
整体视图
业务拓扑
趋势图-响应时间
800
600
400
200
0
390ms
109ms
18ms
0ms
09:4709:5009:5209:5409:5709:5910:01
04/10 10:04
10:11 10:13 10:15 10:18 10:20 10:22
电商平台-响应时间
订单系统-响应时间
服务器1-响应时间
网站主机-响应时间
业务拓扑-电子商务系统
资源使用一览
500
400
300
200
100
0
电商平台
服务器1
订单系统
网站主机
url百度
趋势图
800
600
400
200
0
09:4709:5009:5209:5409:5709:5910:01
订单系统-响应时间
电商平台-响应时间

淘宝双11效果图

Storm计算架构特点

- 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

Storm计算流程


水源头
水龙头
转接头
处理后的水存放池
6）Topology：Storm中运行的一个实时应用程序的名称
DataSource
Spout
tuple
Bolt
tuple
Bolt
hdfs
5）Stream是指数据流
tuple
Bolt
tuple
Hbase
1）文件、数据库、缓冲队列kafka等
4）tuple是一次消息传递的基本单元.
Mongodb
DataSource
Spout
tuple
Bolt
Redis
2）spout从外部数据源中读取数据，然后转换为topology内部的源数据
3）bolt接收数据后，根据用户需求执行相应的操作
知乎 @Gavin

Storm实时流计算架构

- Flume获取数据
- Kafka临时保存数据
- Strom计算数据
- Redis是个内存数据库，用来保存数据

1. 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：

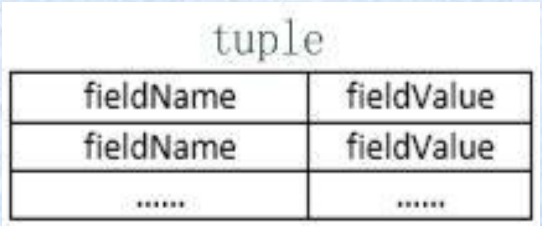
- 数据模型 Tuple
- 数据流 Stream
- 数据源 Spout
- 处理单元 Bolt
- 分发策略 Stream Grouping
- 逻辑视图 Topology

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

Tuple格式

Tuple数据结构



Tuple值域支持私有类型、字符串、字节数组等作为它的字段值，如果使用其他类型，就需要序列化该类型。

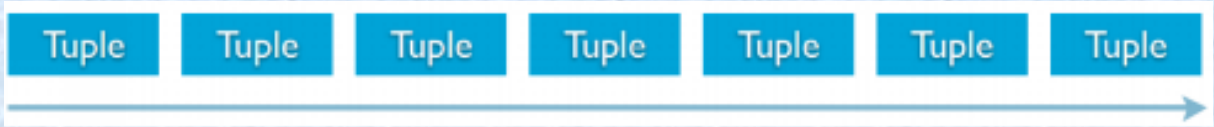
Tuple的字段默认类型有： integer、 float、 double、 long、short、string、 byte、 binary（byte[ ]）

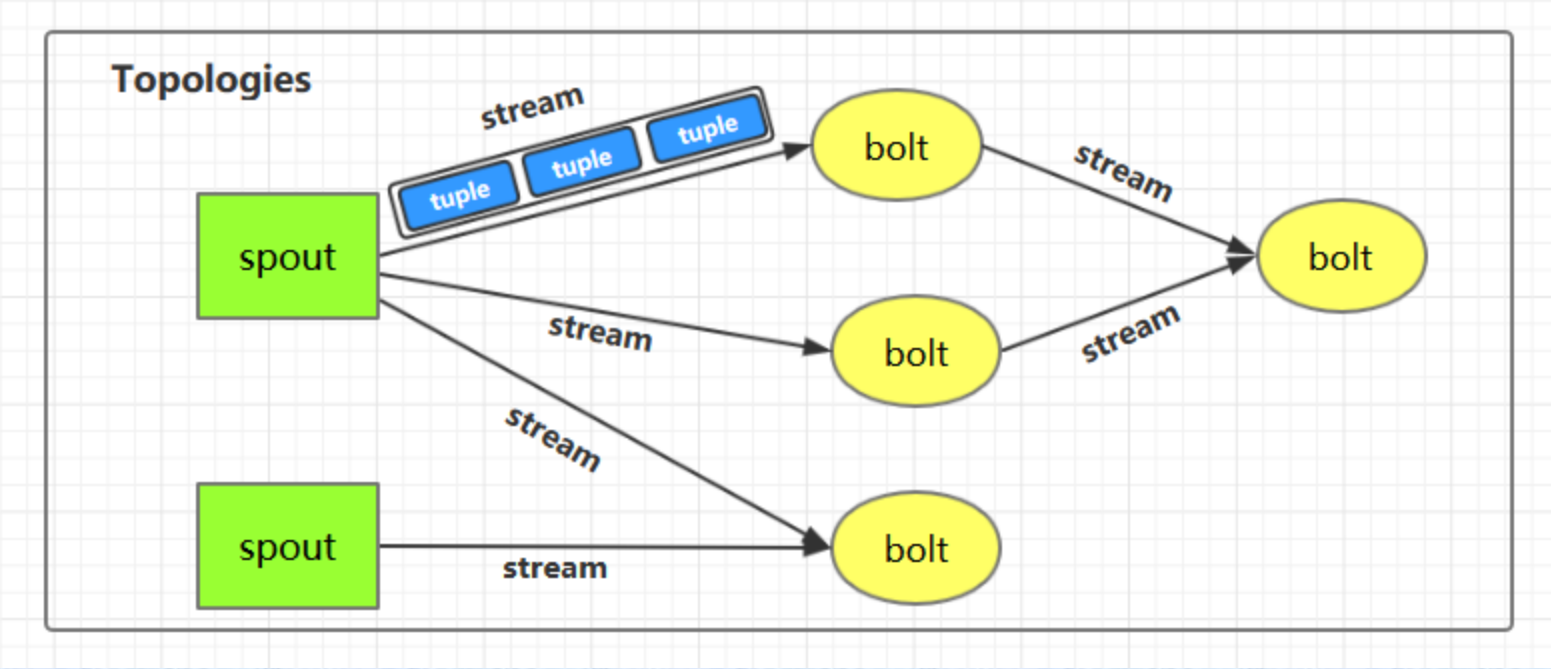
```
public class DoubleAndTripleBolt extends BaseRichBolt {
OutputCollectorBase _collector;
@Override
public void prepare(Map conf, TopologyContext context, OutputCollectorBase
collector) {
_collector = collector;
}
@Override
public void execute(Tuple input) {
int val = input.getInteger(0);
_collector.emit(input, new Values(val*2,  val*3));
_collector.ack(input);
}
@Override
public void declareOutputFields(OutputFieldsDeclarer declarer) {
declarer.declare(new Fields("double", "triple"));
}
}
```

数据流Stream

Stream是一个不间断的无界的连续Tuple序列，是对流数据的抽象

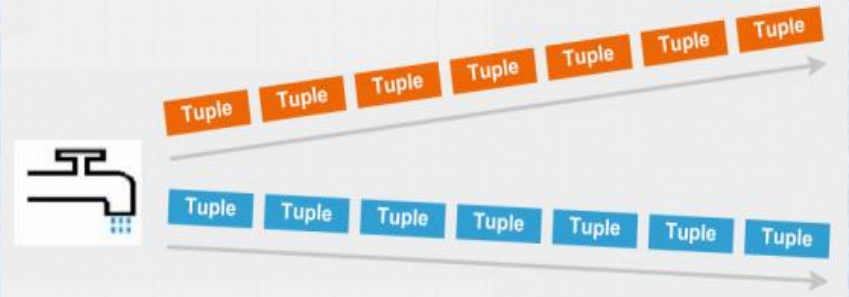
Stream组成

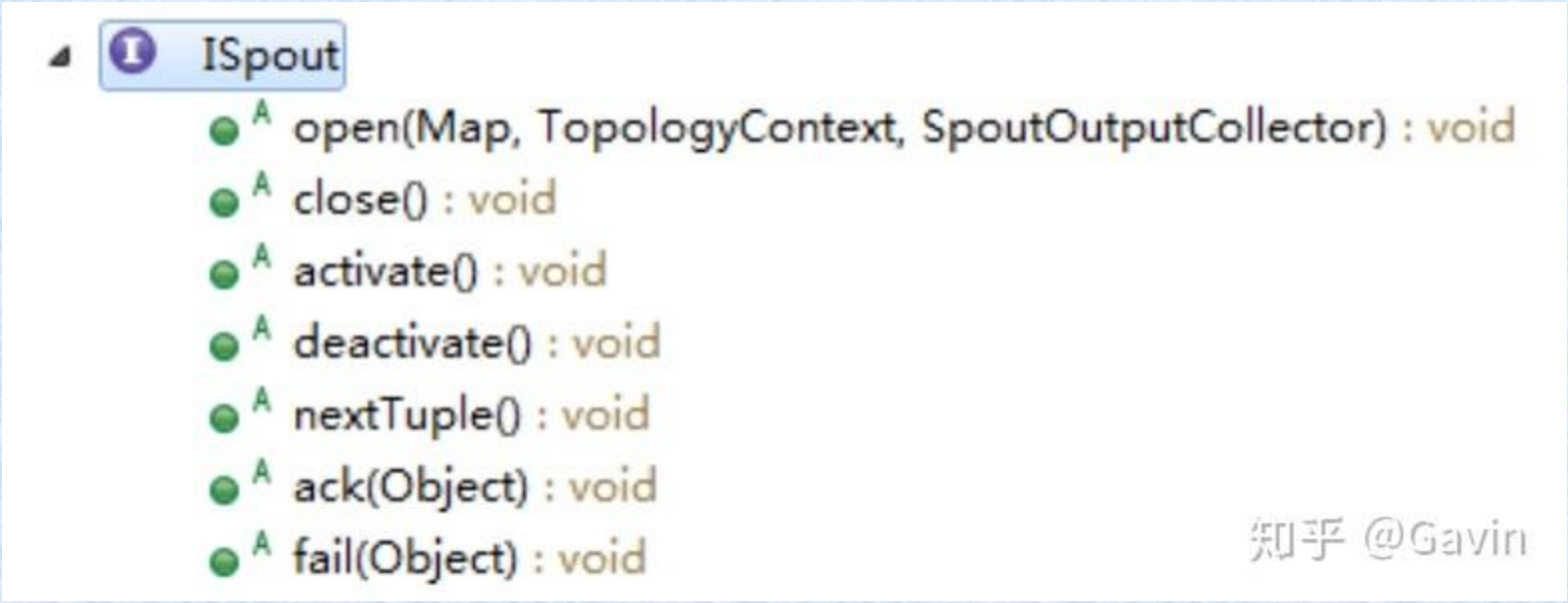




数据源Spout： 负责将外部输入数据流转换成Tuple序列

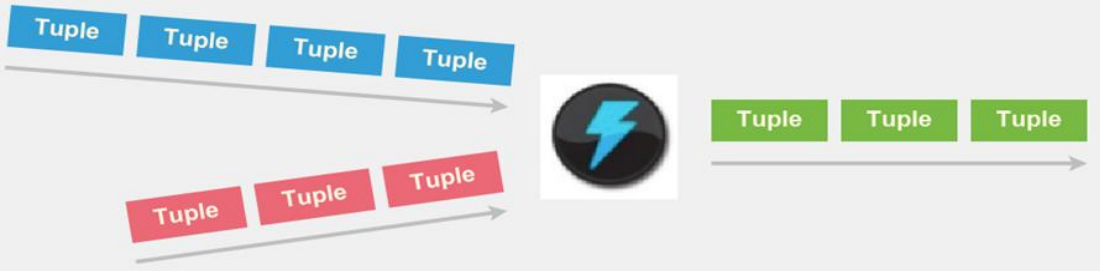
数据源Spout





处理单元Bolt： Bolt将所有的消息处理逻辑都封装在执行程序里面，可执行过滤、聚合、查询数据库等操作，它接收输入的Tuple流并产生输出的新Tuple流

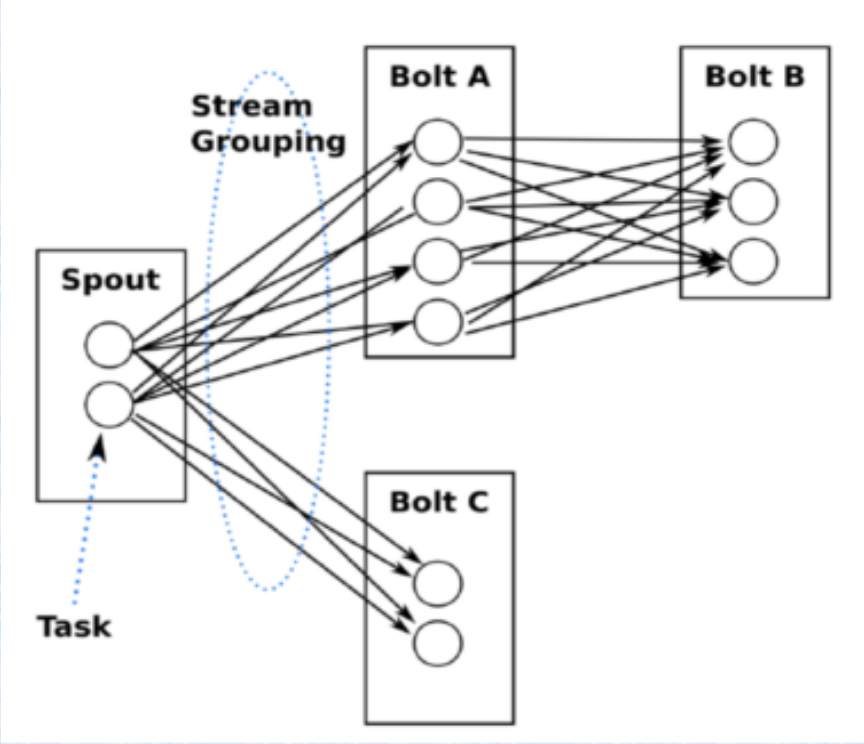
处理单元Bolt



消息分发策略 Stream Grouping

Tuple序列从上游Bolt到下游Bolt其多个并发Task的分组分发方式。

- 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



分发策略Stream Grouping

逻辑视图 Topology（逻辑架构）

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。

- ✓ 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。

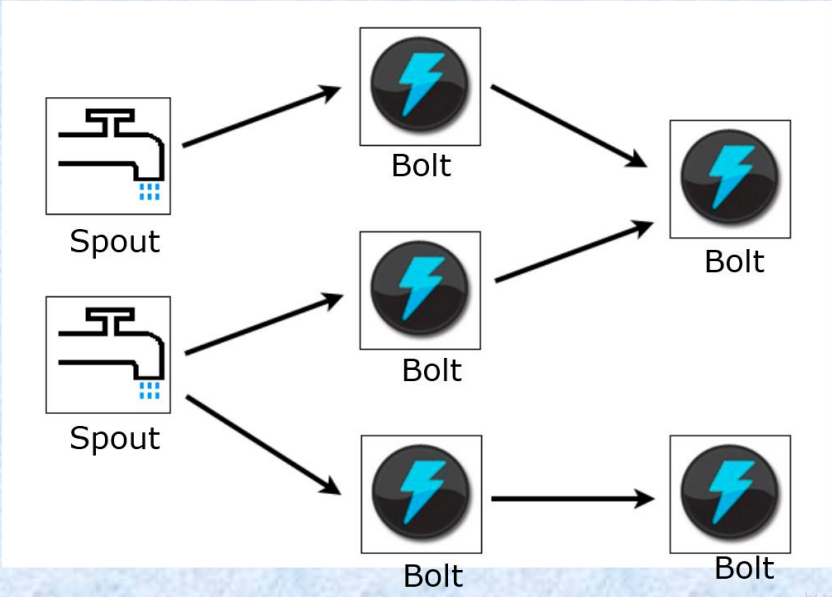
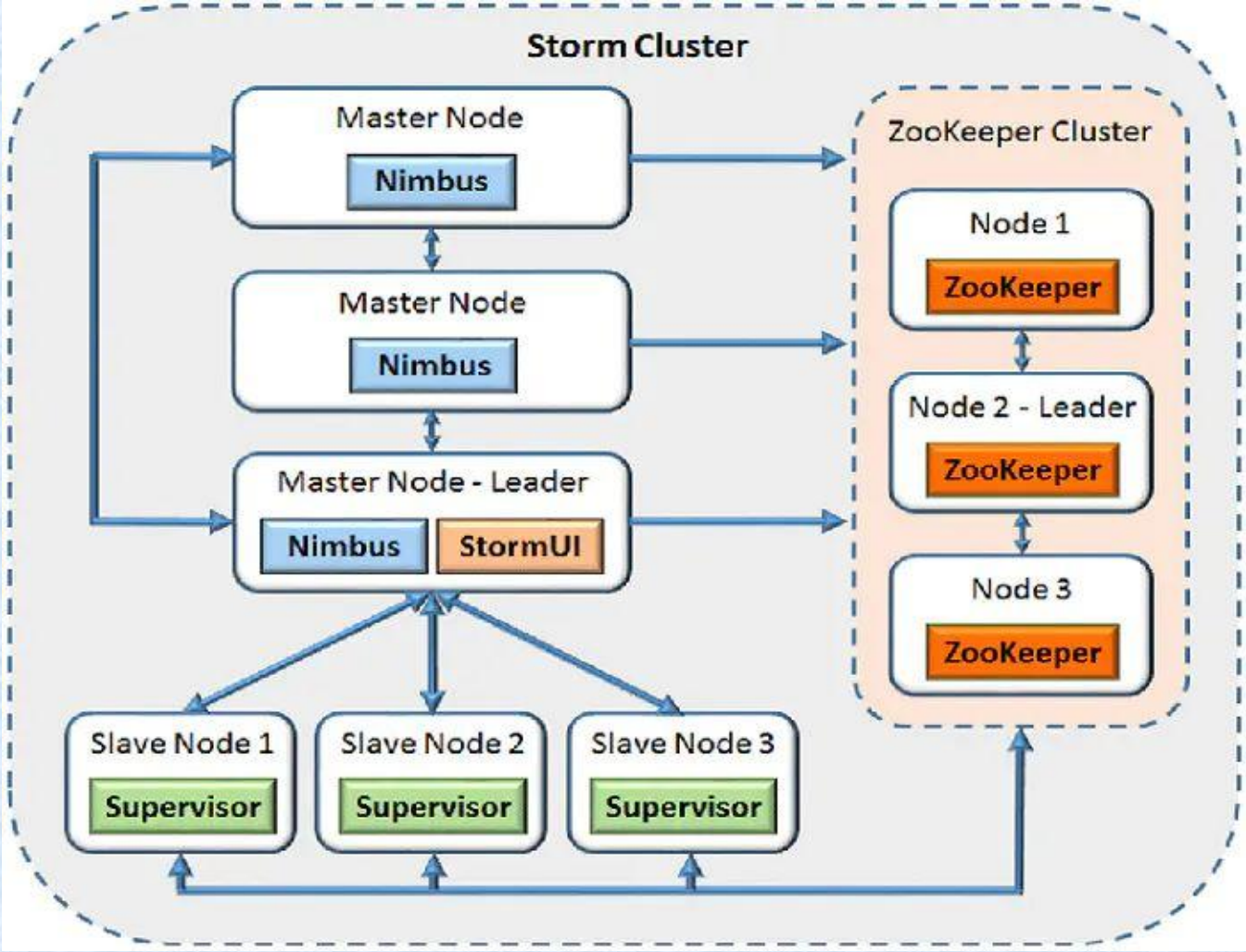
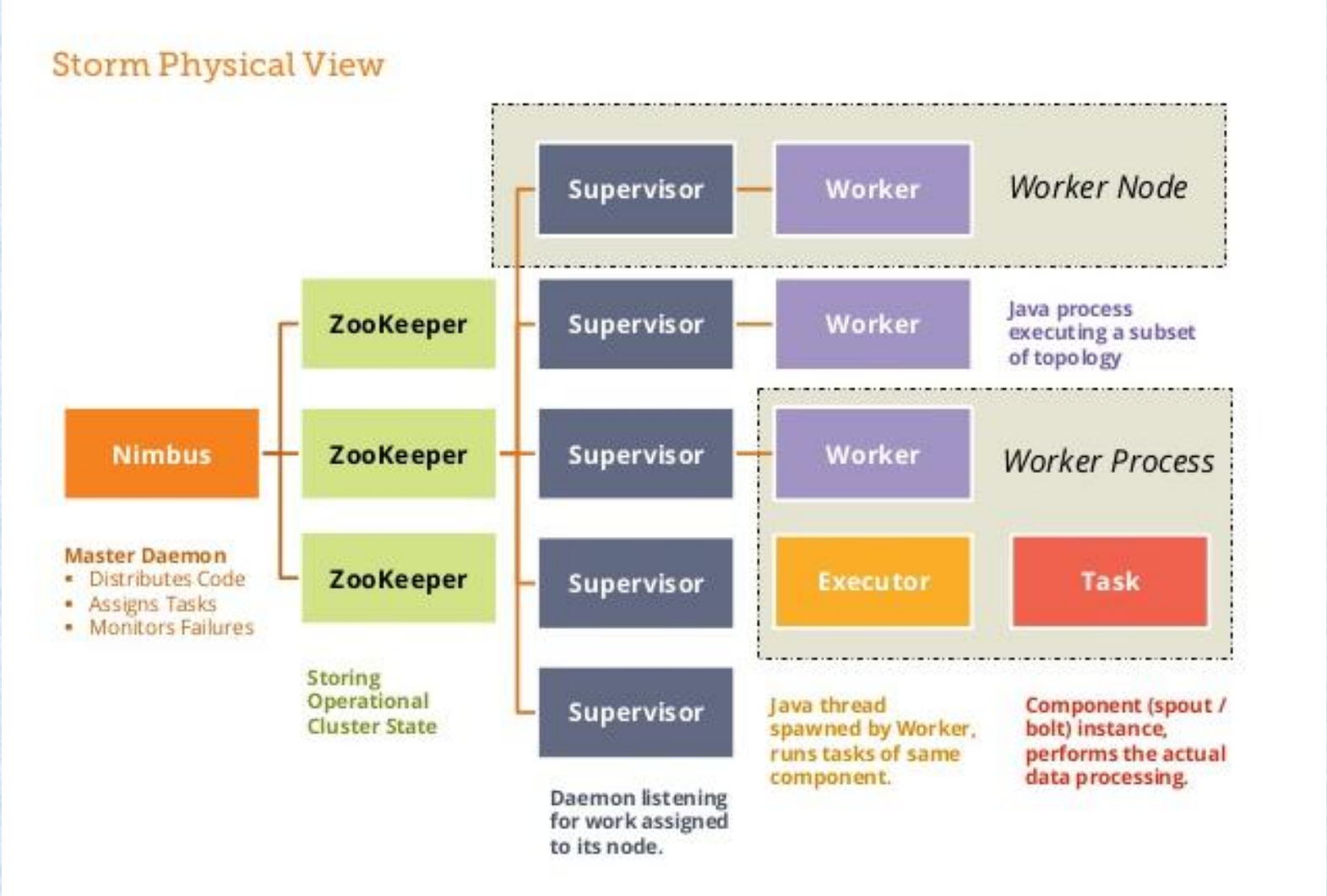


图 15-23　Topology 视图

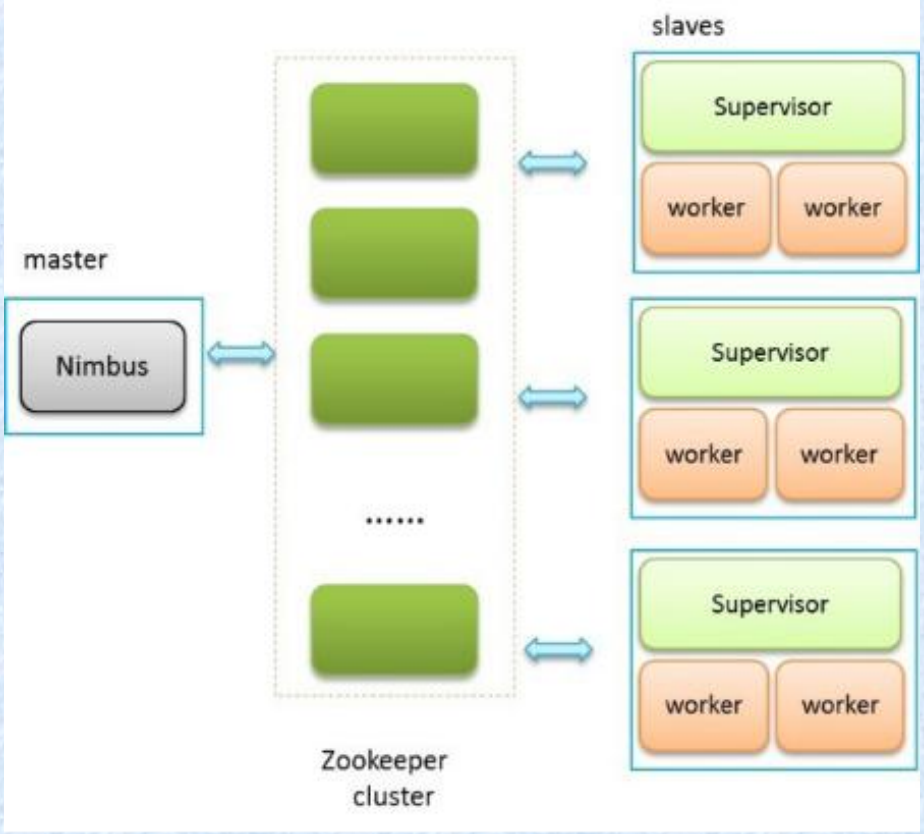
2. 系统架构


Storm Cluster
Master Node
Nimbus
Master Node
Nimbus
Master Node - Leader
Nimbus
StormUI
ZooKeeper Cluster
Node 1
ZooKeeper
Node 2 - Leader
ZooKeeper
Node 3
ZooKeeper
Slave Node 1
Supervisor
Slave Node 2
Supervisor
Slave Node 3
Supervisor


Storm Physical View
Supervisor
Worker
Worker Node
ZooKeeper
Supervisor
Worker
Java process executing a subset of topology
Nimbus
ZooKeeper
Supervisor
Worker
Worker Process
Master Daemon
• Distributes Code
• Assigns Tasks
• Monitors Failures
ZooKeeper
Supervisor
Executor
Task
Storing Operational Cluster State
Supervisor
Java thread spawned by Worker, runs tasks of same component.
Component (spout / bolt) instance, performs the actual data processing.
Daemon listening for work assigned to its node.

物理架构

- Storm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Slave
- 主节点上运行一个Nimbus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管理众多工作节点
- 每个工作节点运行一个Supervisor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。



Storm的系统架构（物理架构）组件

● Storm主控程序	Nimbus	物理计算单元
● 集群调度器	Zookeeper	物理计算单元
● 工作节点控制程序	Supervisor	物理计算单元
● 工作进程	Worker	逻辑计算单元
● 执行进程	Executor	物理计算单元
● 计算任务	Task	逻辑计算单元

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。

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。

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。

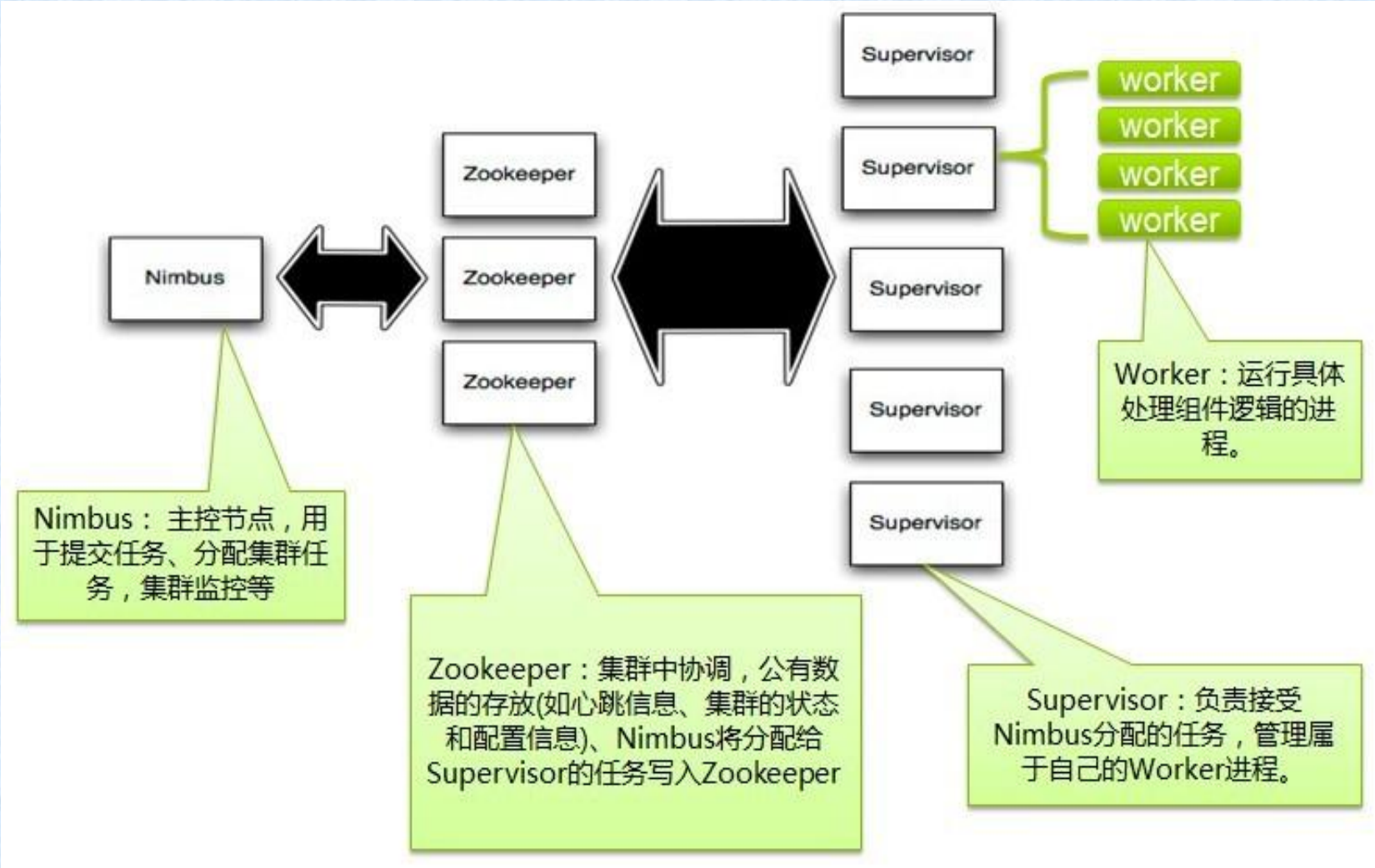
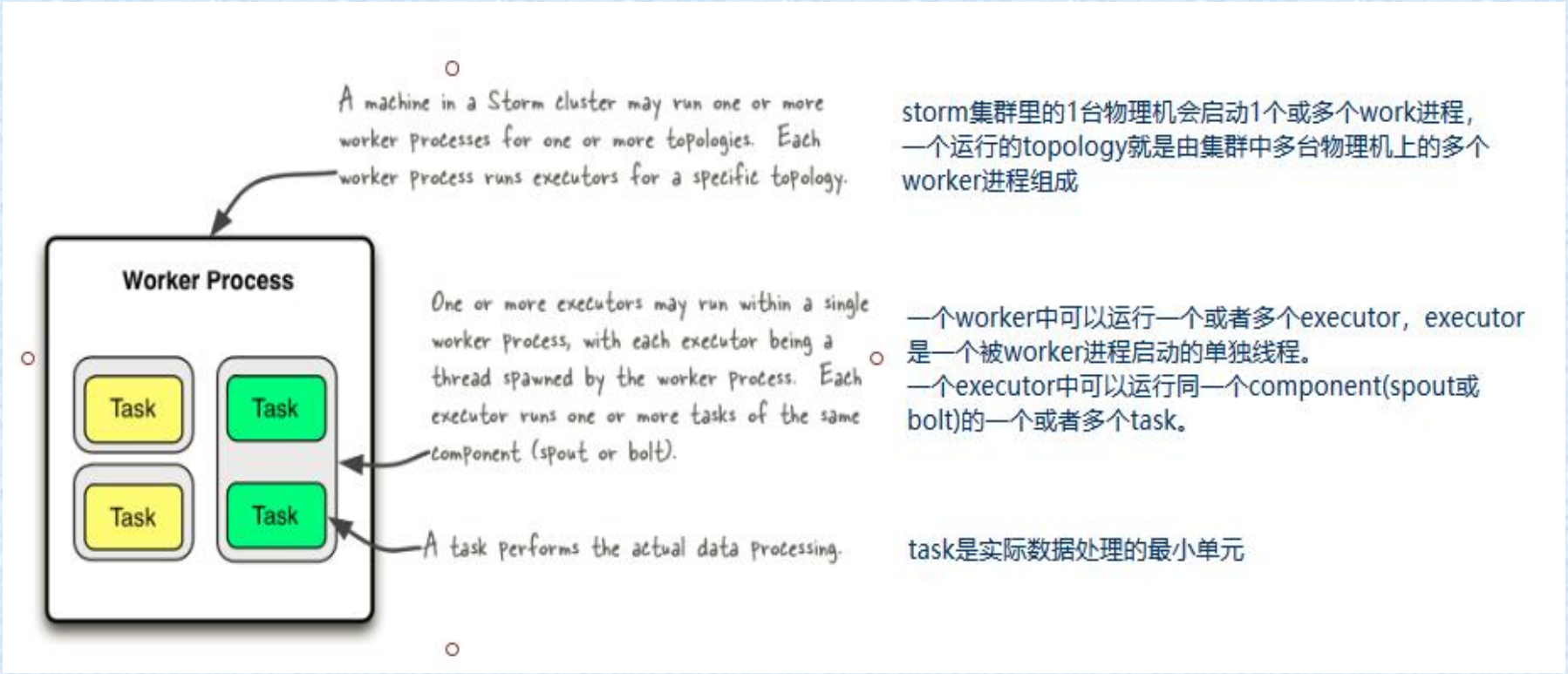


图 15-25　Storm 的技术架构

- 一个运行中的 Topology 由集群中的多个 Worker 进程组成的
- 在默认情况下，每个 Worker 进程默认启动一个 Executor 线程
- 在默认情况下，每个 Executor 默认启动一个 Task 线程；
- Task 是组成 Component 的代码单元



- 1 个 Worker 进程执行的是 1 个 Topology 的子集，不会出现 1 个 Worker 为多个 Topology 服务的情况，因此 1 个运行中的 Topology 就是由集群中多台物理机上的多个 Worker 进程组成的。
- 1 个 Worker 进程会启动 1 个或多个 Executor 线程来执行 1 个 Topology 的 Component(组件，即 Spout 或 Bolt)
- Executor 是 1 个被 Worker 进程启动的单独线程。每个 Executor 会运行 1 个 Component 中的一个或者多个 Task
- Task 是组成 Component 的代码单元。Topology 启动后，1 个 Component 的 Task 数目是固定不变的，但该 Component 使用的 Executor 线程数可以动态调整（例如：1 个 Executor 线程可以执行该 Component 的 1 个或多个 Task 实例）。这意味着，对于 1 个 Component 来说，#threads<=#tasks（线程数小于等于 Task 数目）这样的情况是存在的。默认情况下 Task 的数目等于 Executor 线程数，即 1 个 Executor 线程只运行 1 个 Task。

3. Storm工作机制

Topology提交与执行

Storm作业Topology的提交过程如图所示。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failed，Nimbus会把它重新分配到其它节点。

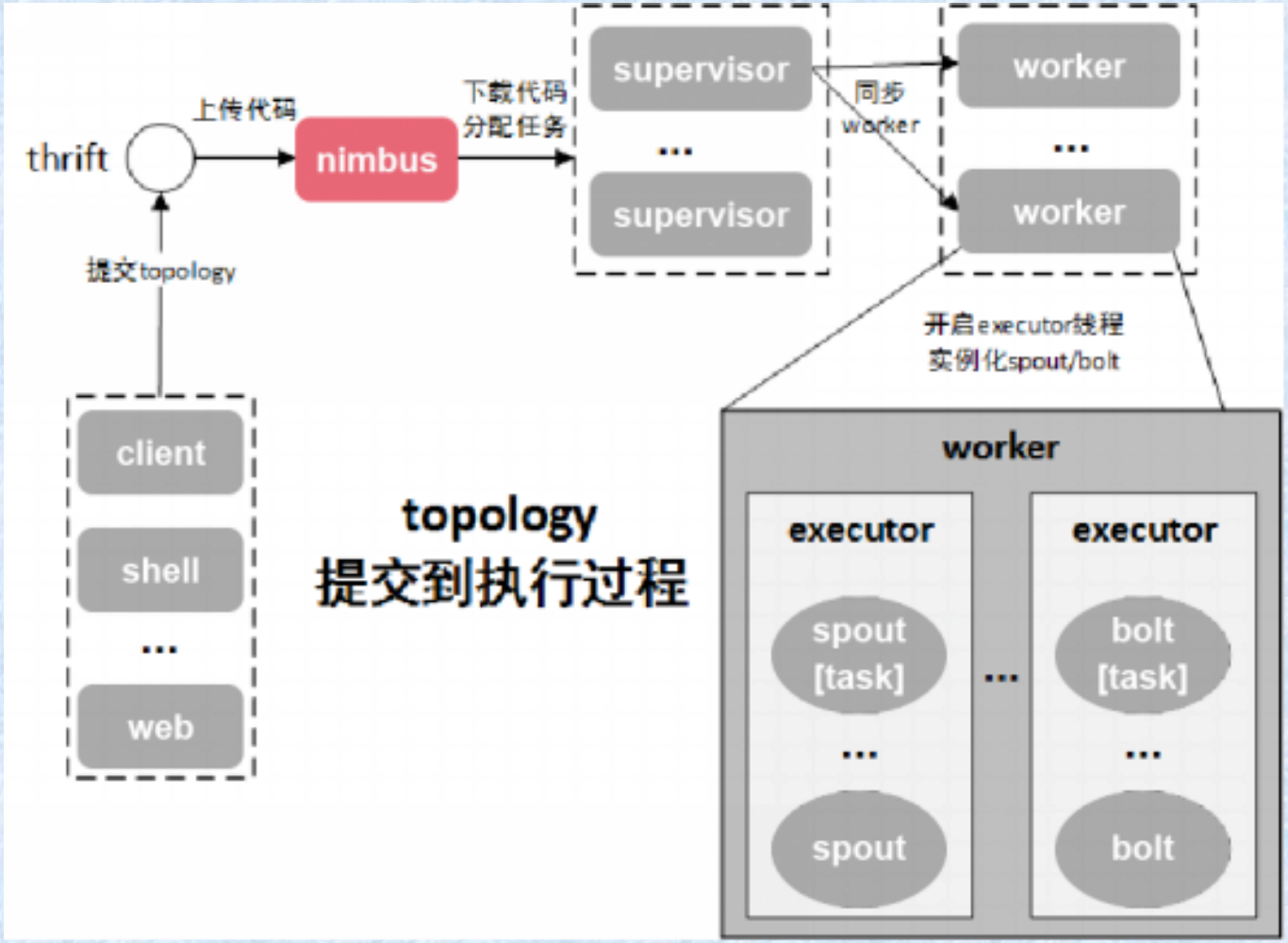
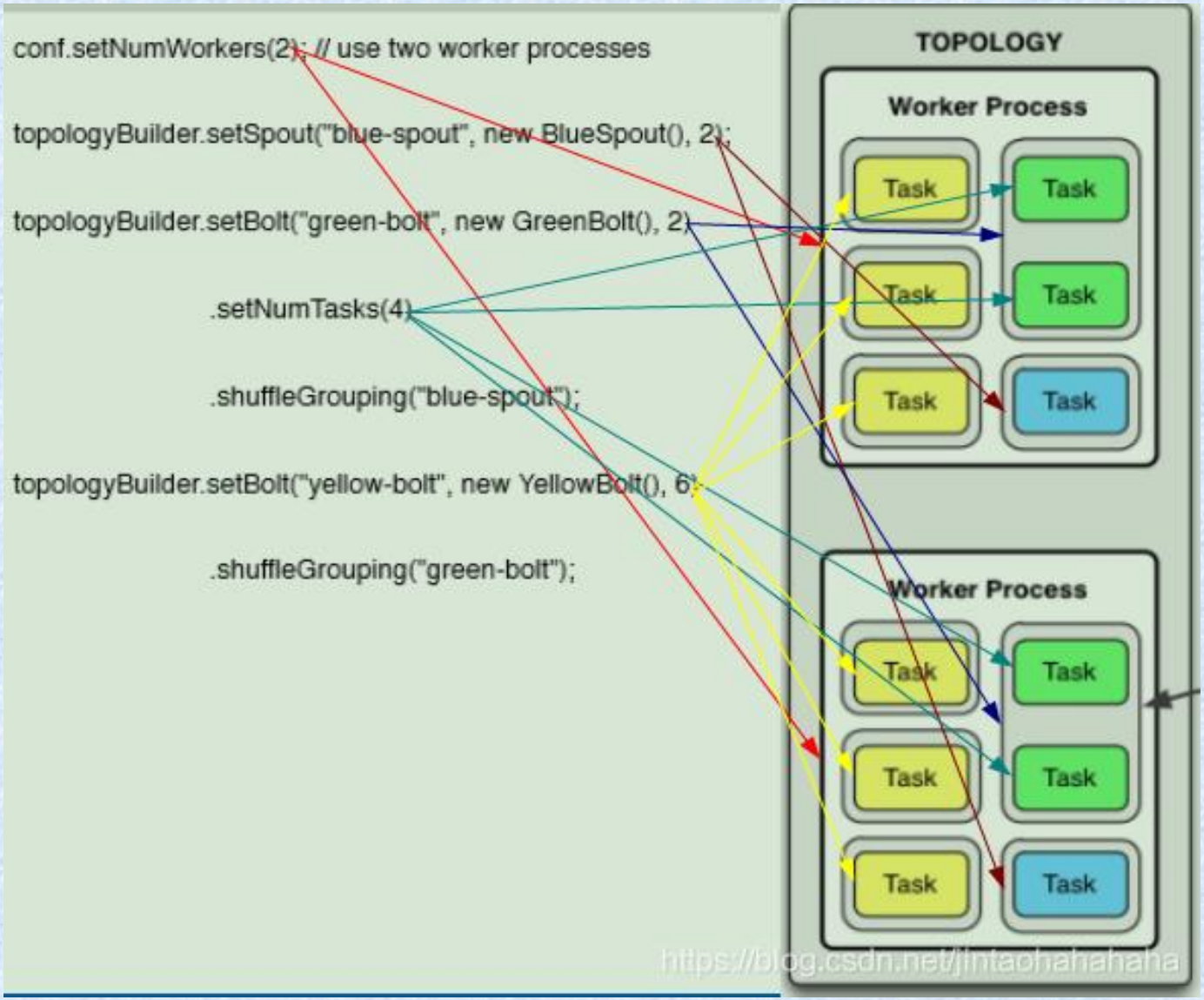


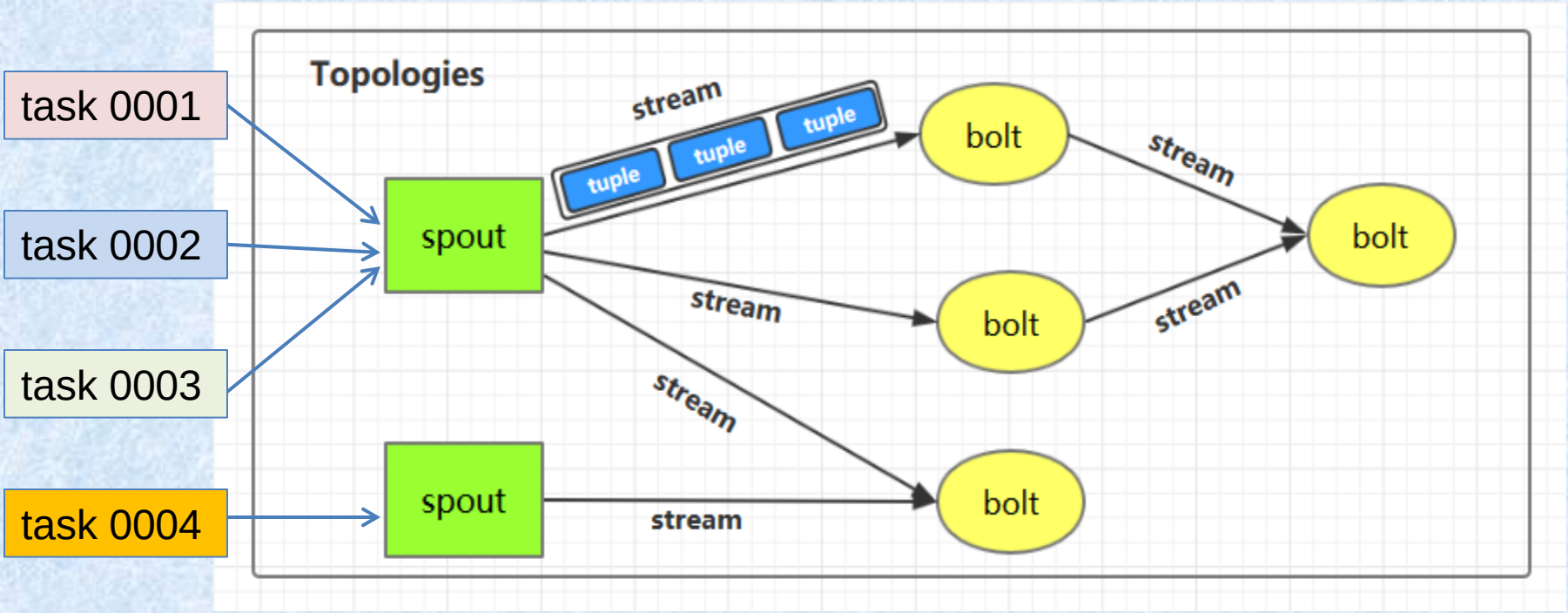
图 15-27　Topology 提交执行过程


conf.setNumWorkers(2); // use two worker processes
topologyBuilder.setSpout("blue-spout", new BlueSpout(), 2);
topologyBuilder.setBolt("green-bolt", new GreenBolt(), 2)
.setNumTasks(4)
.shuffleGroupingblue-spout");
topologyBuilder.setBolt("yellow-bolt", new YellowBolt(), 6)
.shuffleGrouping("green-bolt");
TOPOLOGY
Worker Process
Task
Task
Task
Task
Task
Task
Worker Process
Task
Task
Task
Task
Task
Task
https://blog.csdn.net/jintaohahahaha

Storm的ACK机制需要解决的问题

- 跟踪Spout发出的每一条tuple stream状态，判断每一个tuple在每一个Bolt处是否成功完成处理
- 能够判断一条tuple stream结束，通知发起这条tuple stream的对应task
- 如果某条tuple stream处理失败或超时，提供处理方法
- Acker运行在内存中（可以有多个Acker并发），但不能负荷太重，消耗内存（不可能为跟踪每个tuple分配一个内存空间）

Task, Spout and Acker的并发



哈希映射 Acker_id = Spout_id MOD(#Ackers)

Acker01: spout_1 (task 0001, task 0002, task 0003)
Acker02: spout_2 (task 0004)

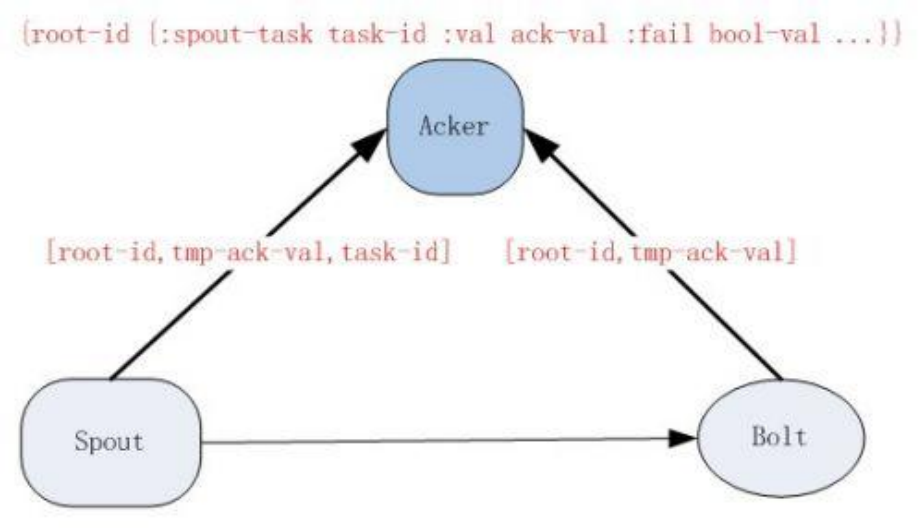
Acker消息机制

Acker跟踪算法3个环节：

1）Spout创建新tuple的时候会给Acker发送消息

2）Bolt中的tuple被ack的时候给Acker发送消息

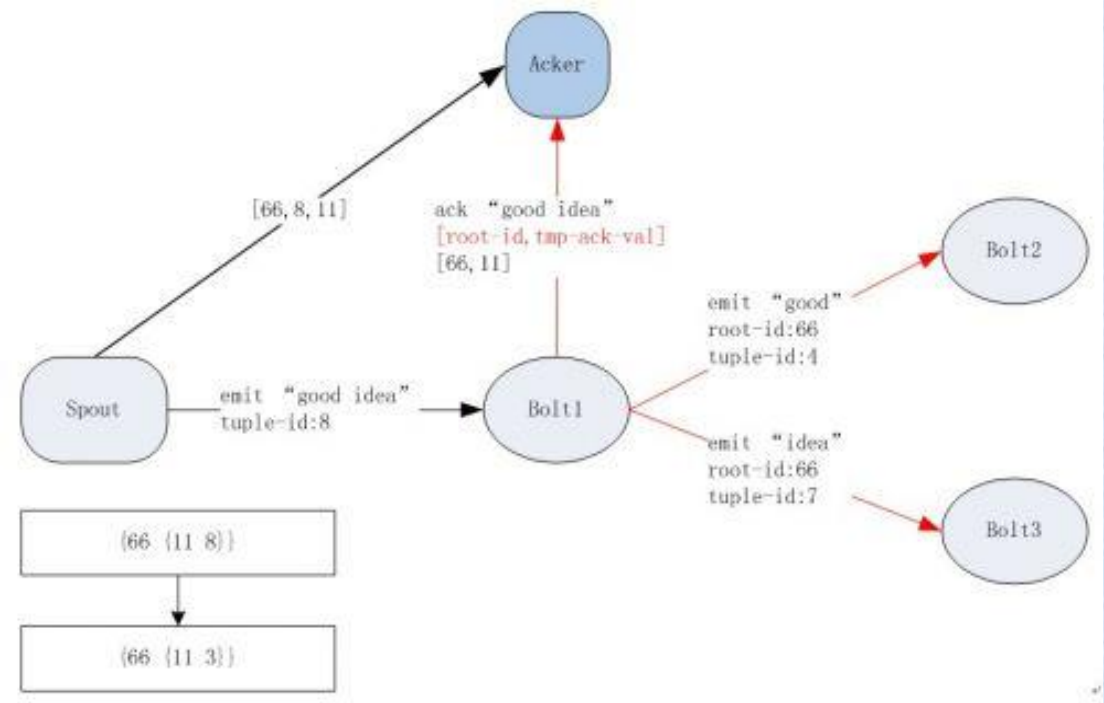
3）Acker跟踪每一个tuple stream, 根据接收到的消息做bitwise XOR运算，更新自己的ack-val



Acker跟踪算法维护这样一个数据结构：

{root-id {:spout-task task-id :val ack-val :failed bool-val …} }

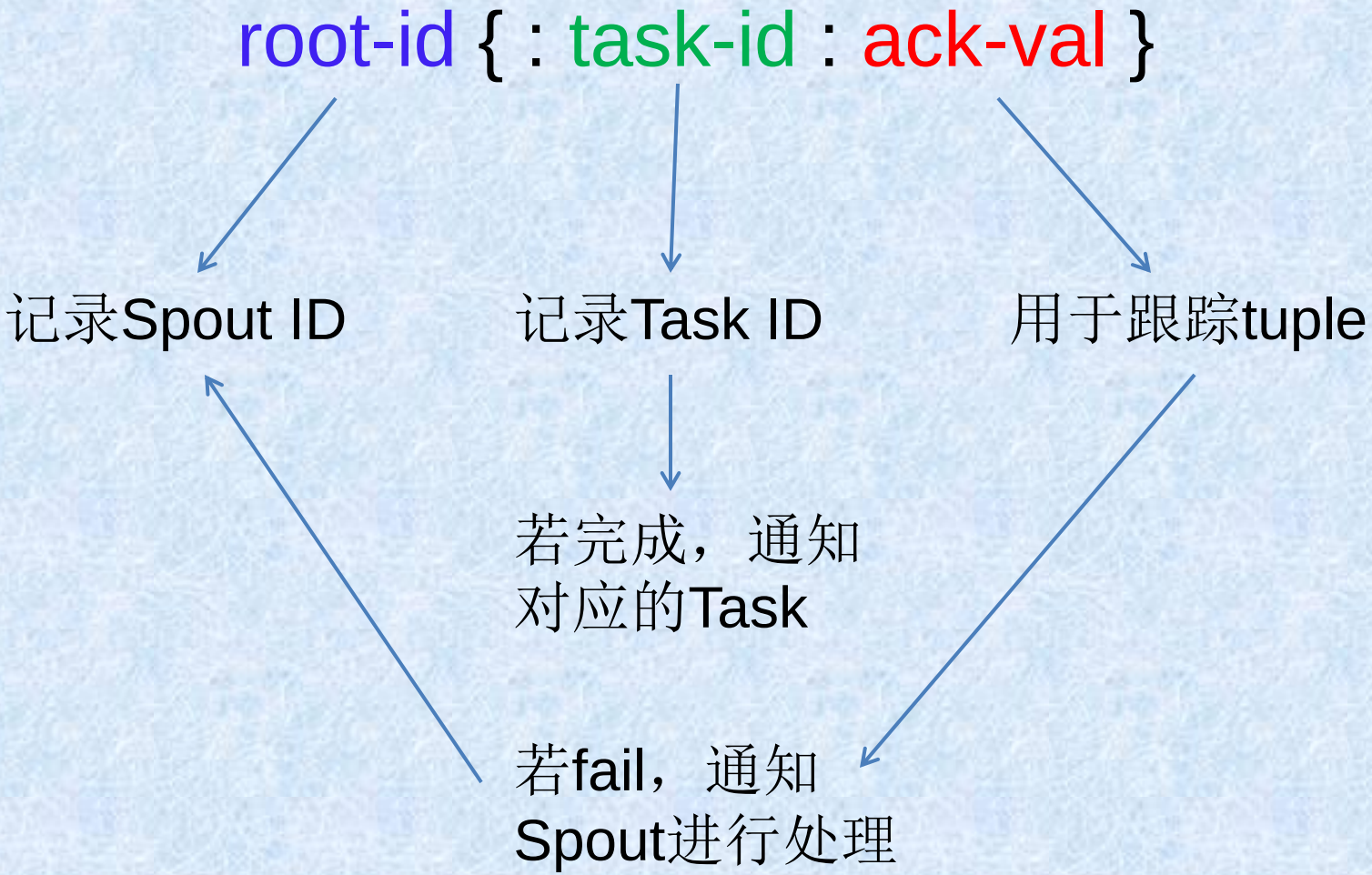
Acker收到的数值



Spout发送给Acker: { root-id { : task-id : ack-val } } { 66, { 8, 11 } }

Bolt发送给Acker: {root-id : ack-val } { 66, 11 }

Acker的计数器


root-id { : task-id : ack-val }
记录Spout ID
记录Task ID
用于跟踪tuple
若完成，通知
对应的Task
若fail，通知
Spout进行处理

Tuple Tree的构成

■ 前(parent)、后(child) tuples的锚定（anchor）

Spout发出的tuple都带有一个64-bit随机生成的 msgId

SpoutOutputCollector.emit (new Values("value1","value2"), msgId);

当Bolt向下游输出衍生的tuple时，调用如下方法建立起输入tuple和输出tuple的关联关系，这称之为锚定（anchor）：

BoltOutputCollector.emit (in-tuple, new Values(word)); //anchor word to in-tuple

■ emit()建立的tuple关联关系在跟踪这个tuple的Acker那里会构成一张DAG图。Bolt接收输入tuple进行处理，处理成功则向Acker发送Ack确认；失败则发送fail报错。这样Acker可以跟踪这张Tuple Tree图里每一个tuple的完成状态。

消息发送ACK机制

Storm可靠性要求发出的每一个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的状态跟踪

以如图的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。

Storm作业的每一个Topology中都包含一个Acker组件。Acker的任务就是跟踪从Spout发出的每一个tuple及其子tuples的处理完成情况，实际上Acker是以一种特殊Task运行，可以通过Config.setNumAckers (conf, ackerParal)设置Acker Task的数目大于1（默认是1），

Acker还可用于Spout限流作用：为了避免Spout发送数据太快而Bolt来不及处理。当Spout有等于或超过pending值的tuples没有收到Ack或fail了，则Spout跳过nextTuple()方法不生成下一个新tuple，从而限制Spout的发送速度。

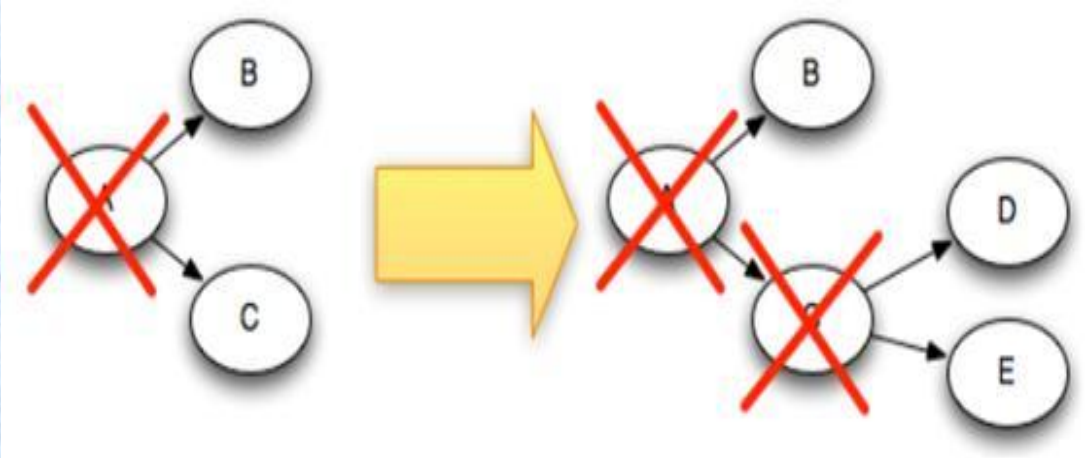


图 15-29　Tuple Tree 的更新

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

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

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

task-id是产生这条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的校验值计数器，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（下表），并将结果更新Acker所持的目前ack-val值：

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

Acker算法（续）

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

Acker算法（续）

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。

Acker算例

下面我们以下图的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。

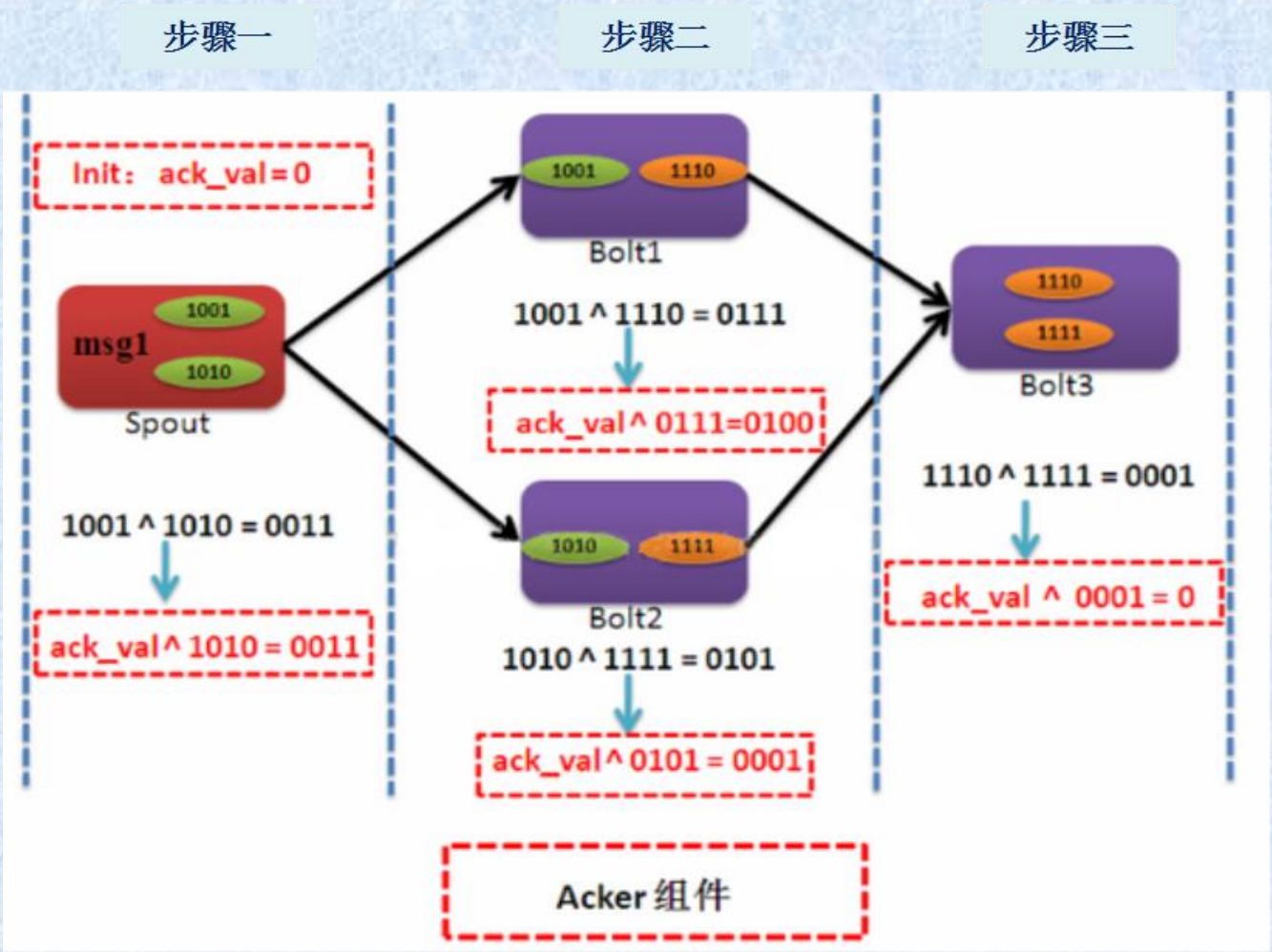


图 15-30　Acker 算法算例

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。

容错算法

- 由于对应的task挂掉了，一个tuple没有被ack：storm的超时机制在超时之后会把这个tuple标记为失败，从而可以重新处理。
- Acker挂掉了： 这种情况下由这个acker所跟踪的所有spout tuple都会超时，也就会被重新处理。
- Spout挂掉了： 在这种情况下给spout发送消息的消息源负责重新发送这些消息。比如Kestrel和RabbitMQ在一个客户端断开之后会把所有”处理中“的消息放回队列。

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-level）

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。

2. Bolt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。

WordCount算例