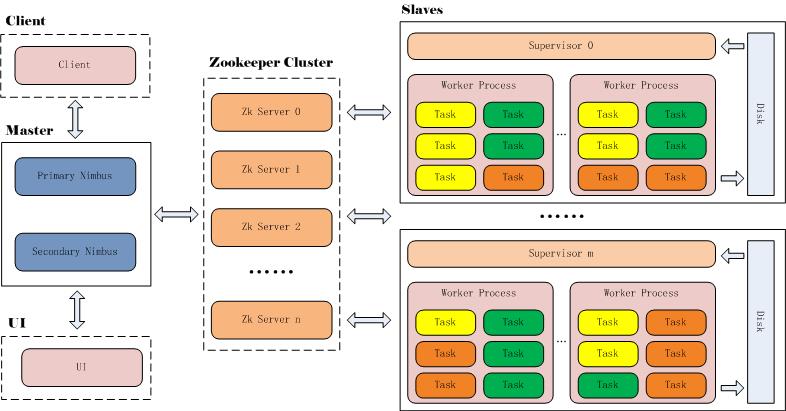
**Jstorm简介**

**整体架构**

物理架构



JStorm保持了Master/Slave的简洁优雅架构。JStorm的M/S之间不是直接通过RPC交换心跳信息，而是借助ZK来实现，这样的设计虽然引入了第三方依赖，但是简化了Nimbus/Supervisor的设计，同时也极大提高了系统的容错能力。

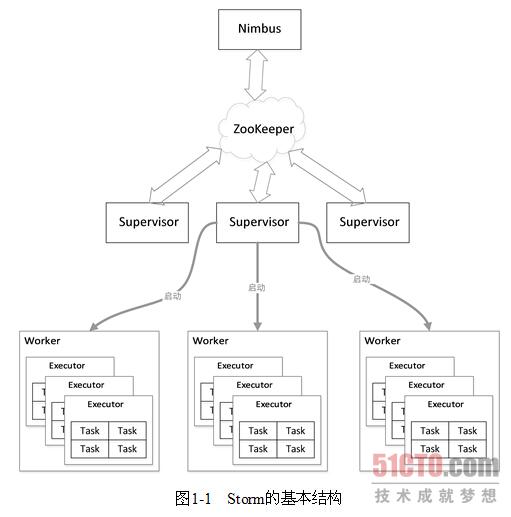
整个JStorm系统中共存三类不同的Daemon进程，分别是Nimbus，Supervisor和Worker。

Nimbus：JStorm中的主控节点，负责接收和验证客户端提交的Topology，分配任务，向ZK写入任务相关的元信息，此外，Nimbus还负责通过ZK来监控节点和任务健康情况，当有Supervisor节点变化或者Worker进程出现问题时及时进行任务重新分配。Nimbus分配任务的结果不是直接下发给Supervisor，也是通过ZK维护分配数据进行过渡。特别地，JStorm 0.9.0领先Apache Storm实现了Nimbus HA，由于Nimbus是Stateless节点，所有的状态信息都交由ZK托管，所以HA相对比较简单，热备Nimbus subscribe ZK关于Master活跃状态数据，一旦发现Master出现问题即从ZK里恢复数据后可以立即接管。

Supervisor：JStorm中的工作节点，Supervisor类似于MR的TT，subscribe ZK分配到该节点的任务数据，根据Nimbus的任务分配情况启动/停止工作进程Worker。Supervisor需要定期向ZK写入活跃端口信息以便Nimbus及时监控。Supervisor不执行具体的数据处理工作，所有的数据处理工作都交给Worker完成。

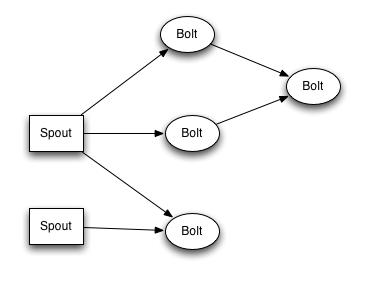
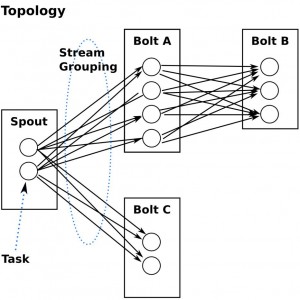
Worker：JStorm中任务执行者，Worker类似于MR的Task，所有实际的数据处理工作最后都在Worker内执行完成。Worker需要定期向Supervsior汇报心跳，由于在同一节点，同时为保持节点的无状态，Worker定期将状态信息写入本地磁盘，Supervisor通过读本地磁盘状态信息完成心跳交互过程。Worker绑定一个独立端口，Worker内所有单元共享Worker的通信能力。

Nimbus、Supervisor和Worker均为Stateless节点，支持Fail-Fast，这为JStorm的扩展性和容错能力提供了很好的保障。



(此部分与Storm一致)

逻辑架构

相关概念和流程（storm）

Spout：JStorm的消息源。用于生产消息，一般是从外部数据源（如MQ/RDBMS/NoSQL/RTLog等）不间断读取数据并向下游发送消息。

Bolt：JStorm的消息处理者。用于为Topology进行消息处理，Bolt可以执行查询、过滤、聚合及各种复杂运算操作，Bolt的消息处理结果可以作为下游Bolt的输入不断迭代。

Stream：JStorm中对数据进行的抽象，它是时间上无界的Tuple元组序列。在Topology中Spout是Stream的源头，负责从特定数据源发射Stream；Bolt可以接收任意多个Stream输入然后进行数据的加工处理，如果需要Bolt还可以发射出新Stream给下游Bolt。

Worker是工作进程。一个工作进程中可以含有一个或多个Executor线程。

Executor是运行Spout或Bolt处理逻辑的线程。

Task是Storm中的最小处理单元。一个Executor中可以包含一个或多个Task，消息的分发都是从一个Task到另一个Task进行的。

Stream Grouping定义了消息分发策略，定义了Bolt节点以何种方式接收数据。有以下几类：

0) Shuffle Grouping：随机分组，随机派发Stream里面的Tuple，保证每个Bolt接收到的Tuple数目大致相同，通过轮询随机的方式使得下游Bolt之间接收到的Tuple数目差值不超过1。  
 1) Fields Grouping：按字段分组，具有同样字段值的Tuple会被分到相同Bolt里的Task，不同字段值则会被分配到不同Task。  
 2) All Grouping：广播分组，每一个Tuple，所有的Bolt都会收到。  
 3) Global Grouping：全局分组，Tuple被分配到Bolt中ID值最低的的一个Task。  
 4) Non Grouping：不分组，Tuple会按照完全随机的方式分发到下游Bolt。  
 5) Direct Grouping：直接分组，Tuple需要指定由Bolt的哪个Task接收。 只有被声明为Direct Stream的消息流可以声明这种分组方法。  
 6) Local or Shuffle Grouping：基本同Shuffle Grouping。  
 7) Custom Grouping：用户自定义分组策略，CustomStreamGrouping是自定义分组策略时用户需要实现的接口。

Topology是由消息分组方式连接起来的Spout和Bolt节点网络，它定义了运算处理的拓扑结构，处理的是不断流动的消息。除非杀掉Topology，否则它将永远运行下去。

Tuple：JStorm使用Tuple作为数据模型，存在于任意两个有数据交互的组件（Spout/Bolt）之间。每个Tuple是一组具有各自名称的值，值可以是任何类型，JStorm支持所有的基本类型、字符串以及字节数组，也可以使用自定义类型（需实现对应序列化器）作为值类型。简单来说，Tuple就是一组实现了序列化器带有名称的Java对象集合。

从整个Topology上看，Spout/Bolt可以看作DAG的节点，Stream是连接不同节点之间的有向边，Tuple则是流过Stream的数据集合。

**部分原理**

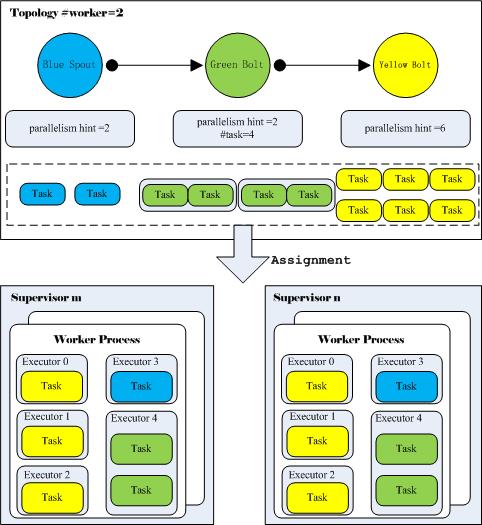
那么Topology的各个计算组件（Spout/Bolt）如何映射到计算资源上。梳理这个问题前需要先明确Worker/Executor/Task之间的关系：

0、Worker：完整的Topology任务是由分布在多个Supervisor节点上的Worker进程（JVM）来执行，每个Worker都执行且仅执行Topology任务的一个子集。

1、Executor：Worker内部会有一个或多个Executor，每个Executor对应一个线程。Executor包括SpoutExecutor和BoltExecutor，同一个Worker里所有的Executor只能属于某一个Topology里的执行单元。

2、Task：执行具体数据处理实体，也就是用户实现的Spout/Blot实例。一个Executor可以对应多个Task，定义Topology时指定，默认Executor和Task一一对应。这就是说，系统中Executor数量一定是小于等于Task数量（Executor≤Task）。

下图给出了一个简单的例子，上半部分描述的是Topology结构及相关说明，其中定义了整个Topology的worker=2，DAG关系，各个计算组件的并行度；下半部分描述了Topology的Task在Supervisor节点的分布情况。从中可以看出Topology到Executor之间的关系。



0、Worker数在提交Topology时在配置文件中指定；

例：#Worker=2

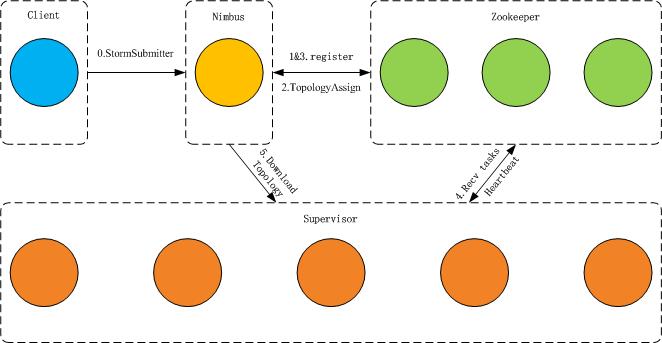
1、执行线程/Executor数在定义Topology的各计算组件并行度时决定，可以不指定，默认为1。其中各个计算组件的并行度之和即为该Topology执行线程总数。

例：#Executor=sum(#parallelism hint)=2+2+6=10

2、Task数目也在定义Toplogy时确定，若不指定默认每个Executor线程对应一个Task，若指定Task数目会在指定数目的线程里平均分配。

例：#Task=sum(#task)=2+4+6=12，其中Executor4={Task0,Task1}

**整体处理流程**



0、Topology提交

JStorm为用户提供了StormSubmitter.submitTopology用来向集群提交Topology，整个提交流程：

Client端：  
0）客户端简单验证；  
1）检查是否已经存在同名Topology；  
2）提交jar包；  
3）向Nimbus提交Topology；

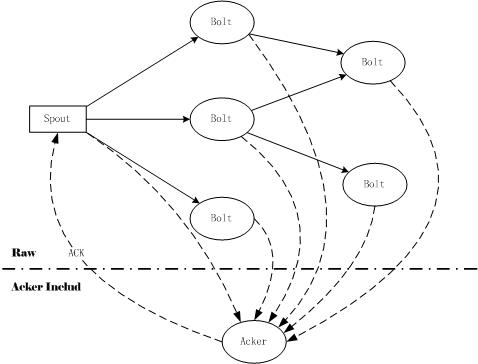
Nimbus端：  
0）Nimbus端简单合法性检查；  
1）生成Topology Name；  
2）序列化配置文件和Topology Code；  
3）Nimbus本地准备运行时所需数据；  
4）向ZK注册Topology和Task；  
5）将Task压入分配队列等待TopologyAssign分配；

1、任务调度策略

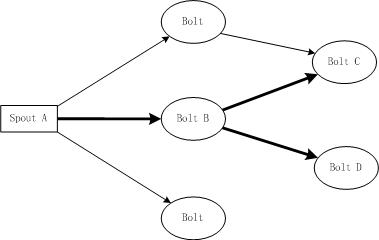
从0.9.0开始，JStorm提供非常强大的调度功能，基本上可以满足大部分的需求，同时支持自定义任务调度策略。JStorm的资源不再仅是Worker的端口，而从CPU/Memory/Disk/Net等四个维度综合考虑。  
Nimbus任务调度算法[2]如下：  
 0）优先使用自定义任务分配算法，当资源无法满足需求时，该任务放到下一级任务分配算法；  
 1）使用历史任务分配算法（如果打开使用历史任务属性），当资源无法满足需求时，该任务放到下一级任务分配算法；  
 2）使用默认资源平衡算法，计算每个Supervisor上剩余资源权值，取权值最高的Supervisor分配任务。

2、Acker机制

为保证无数据丢失，Storm/JStorm使用了非常漂亮的可靠性处理机制，如图当定义Topology时指定Acker，JStorm除了Topology本身任务外，还会启动一组称为Acker的特殊任务，负责跟踪Topolgogy DAG中的每个消息。每当发现一个DAG被成功处理完成，Acker就向创建根消息的Spout任务发送一个Ack信号。Topology中Acker任务的并行度默认parallelism hint=1，当系统中有大量的消息时，应该适当提高Acker任务的并行度。



Acker按照Tuple Tree的方式跟踪消息。当Spout发送一个消息的时候，它就通知对应的Acker一个新的根消息产生了，这时Acker就会创建一个新的Tuple Tree。当Acker发现这棵树被完全处理之后，他就会通知对应的Spout任务。



Acker任务保存了数据结构Map<MessageID,Map< TaskID, Value>>，  
其中MessageID是Spout根消息ID，TaskID是Spout任务ID，Value表示一个64bit的长整型数字，是树中所有消息的随机ID的异或结果。通过TaskID，Acker知道当消息树处理完成后通知哪个Spout任务，通过MessageID，Acker知道属于Spout任务的哪个消息被成功处理完成。Value表示了整棵树的的状态，无论这棵树多大，只需要这个固定大小的数字就可以跟踪整棵树。当消息被创建和被应答的时候都会有相同的MessageID发送过来做异或。当Acker发现一棵树的Value值为0的时候，表明这棵树已经被成功处理完成。

例如，对于前面Topology中消息树，Acker数据的变化过程：  
Step0.A发送T0给B后：  
R0=r0  
<id0,<taskA,R0>>  
Step1.B接收到T0并成功处理后向C发送T1，向D发送T2：  
R1=R0^r1^r2=r0^r1^r2  
<id0,<taskA,R0^R1>>  
=<id0,<taskA,r0^r0^r1^r2>>  
=<id0,<taskA,r1^r2>>  
Step2.C接收到T1并成功处理后：  
R2=r1  
<id0,<taskA,r1^r2^R2>>  
=<id0,<taskA,r1^r2^r1>>  
=<id0,<taskA,r2>>  
Step3.D接收到T2并成功处理后：  
R3=r2  
<id0,<taskA,r2^R3>>  
=<id0,<taskA,r2^r2>>  
=<id0,<taskA,0>>  
当结果为0时Acker可以通知taskA根消息id0的消息树已被成功处理完成。

需要指出的是，Acker并不是必须的，当实际业务可以容忍数据丢失情况下可以不用Acker，对数据丢失零容忍的业务必须打开Acker，另外当系统的消息规模较大是可适当增加Acker的并行度。

3、故障恢复

0）节点故障

Nimbus故障。Nimbus本身无状态，所以Nimbus故障不会影响正在正常运行任务，另外Nimbus HA保证Nimbus故障后可以及时被备份Nimbus接管。  
Supervisors节点故障。Supervisor故障后，Nimbus会将故障节点上的任务迁移到其他可用节点上继续运行，但是Supervisor故障需要外部监控并及时手动重启。  
Worker故障。Worker健康状况监控由Supervisor负责，当Woker出现故障时，Supervisor会及时在本机重试重启。  
Zookeeper节点故障。Zookeeper本身具有很好的故障恢复机制，能保证至少半数以上节点在线就可正常运行，及时修复故障节点即可。

1）任务失败

Spout失败。消息不能被及时被Pull到系统中，造成外部大量消息不能被及时处理，而外部大量计算资源空闲。  
 Bolt失败。消息不能被处理，Acker持有的所有与该Bolt相关的消息反馈值都不能回归到0，最后因为超时最终Spout的fail将被调用。  
 Acker失败。Acker持有的所有反馈信息不管成功与否都不能及时反馈到Spout，最后同样因为超时Spout的fail将被调用。  
任务失败后，需要Nimbus及时监控到并重新分配失败任务。

**五、基础接口**

这里把几个基础接口中注释摘出来说明其的作用：

0、ISpout: ISpout is the core interface for implementing spouts. A Spout is responsible for feeding messages into the topology for processing. For every tuple emitted by a spout, Storm will track the (potentially very large) DAG of tuples generated based on a tuple emitted by the spout. When Storm detects that every tuple in that DAG has been successfully processed, it will send an ack message to the Spout.  
1、IBolt: IBolt represents a component that takes tuples as input and produces tuples as output. An IBolt can do everything from filtering to joining to functions to aggregations. It does not have to process a tuple immediately and may hold onto tuples to process later.  
2、TopologyBuilder: TopologyBuilder exposes the Java API for specifying a topology for Storm to execute.  
3、StormSubmitter: Use this class to submit topologies to run on the Storm cluster.

针对前面例子中的Topology这里给出一个简单的实现，其中略去了BlueSpout/GreeBolt/YellowBolt的具体实现，更多参考这里。

|  |
| --- |
| **public** **static** **void** main ([String](http://www.google.com/search?hl=en&q=allinurl%3Astring+java.sun.com&btnI=I%27m%20Feeling%20Lucky)[] args){  Config conf = **new** Config();  *// use two worker processes*  conf.setNumWorkers(2);  *// set parallelism hint to 2*  topologyBuilder.setSpout("blue-spout", **new** BlueSpout(), 2);  topologyBuilder.setBolt("green-bolt", **new** GreenBolt(), 2)  .setNumTasks(4)  .shuffleGrouping("blue-spout");  topologyBuilder.setBolt("yellow-bolt", **new** YellowBolt(), 6)  .shuffleGrouping("green-bolt");  StormSubmitter.submitTopology(  "mytopology",  conf,  topologyBuilder.createTopology());  } |