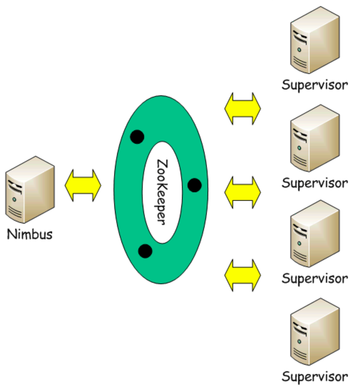
一. Storm集群架构

Storm集群采用主从架构方式，主节点是Nimbus，从节点是Supervisor，有关调度相关的信息存储到ZooKeeper集群中，架构如下图所示：



- **Nimbus**

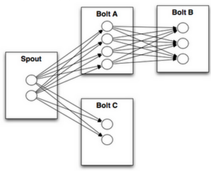
  Storm集群的Master节点，负责分发用户代码，指派给具体的Supervisor节点上的Worker节点，去运行Topology对应的组件（Spout/Bolt）的Task。
- **Supervisor**

  Storm集群的从节点，负责管理运行在Supervisor节点上的每一个Worker进程的启动和终止。通过Storm的配置文件中的supervisor.slots.ports配置项，可以指定在一个Supervisor上最大允许多少个Slot，每个Slot通过端口号来唯一标识，一个端口号对应一个Worker进程（如果该Worker进程被启动）。
- **ZooKeeper**

  用来协调Nimbus和Supervisor，如果Supervisor因故障出现问题而无法运行Topology，Nimbus会第一时间感知到，并重新分配Topology到其它可用的Supervisor上运行。

二. Stream Groupings

Storm中最重要的抽象，应该就是Stream grouping了，它能够控制Spot/Bolt对应的Task以什么样的方式来分发Tuple，将Tuple发射到目的Spot/Bolt对应的Task，如下图所示：

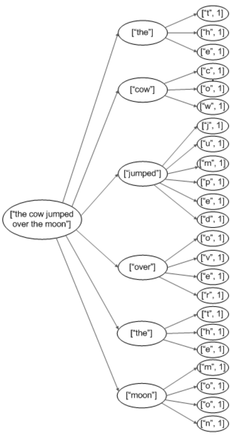


目前，Storm Streaming Grouping支持如下几种类型：

- **Shuffle Grouping**：随机分组，跨多个Task，能够随机使得每个Bolt的Task接收到大致相同数目的Tuple，但是Tuple不重复
- **Fields Grouping**：根据指定的Field进行分组，同一个Field的值一定会被发射到同一个Task上，窗口计算的时候特别有用
- **Partial Key Grouping**：与Fields grouping 类似，根据指定的Field的一部分进行分组分发，能够很好地实现Load balance，将Tuple发送给下游的Bolt对应的Task，特别是在存在数据倾斜的场景，使用 Partial Key grouping能够更好地提高资源利用率
- **All Grouping**：所有Bolt的Task都接收同一个Tuple（这里有复制的含义）
- **Global Grouping**：所有的流都指向一个Bolt的同一个Task（也就是Task ID最小的）
- **None Grouping**：不需要关心Stream如何分组，等价于Shuffle grouping
- **Direct Grouping**：由Tupe的生产者来决定发送给下游的哪一个Bolt的Task，这个要在实际开发编写Bolt代码的逻辑中进行精确控制
- **Local or Shuffle Grouping**：如果目标Bolt有1个或多个Task都在同一个Worker进程对应的JVM实例中，则Tuple只发送给这些Task

三. Acker原理

首先，我们理解一下Tuple Tree的概念，要计算英文句子中每个字母出现的次数，形成的Tuple Tree如下图所示：



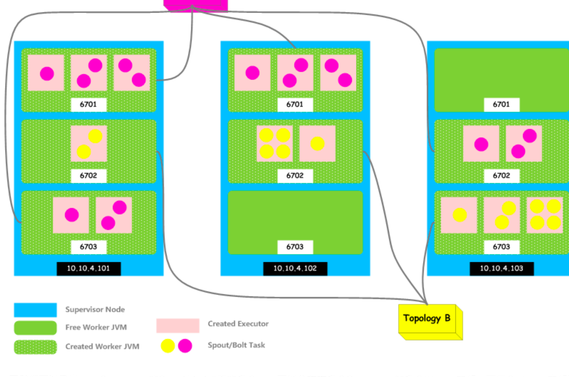
对上述这个例子，也就是说，运行时每一个英文句子都会对应一个Tuple Tree，一个Tuple Tree可能很大，也可能很小，与具体的业务需求有关。另外，Acker也是一个Bolt组件，只不过我们实现处理自己业务逻辑时，不需要关心Acker Bolt的实现，在提交实现的Topology到Storm集群后，会在初始化Topology时系统自动为我们的Topology增加Acker这个Bolt组件，它的主要功能是负责跟踪我们自己实现的Topology中各个Spout/Bolt所处理的Tuple之间的关系（或者可以说是，跟踪Tuple Tree的处理进度）。

下面，我们描述一下Acker的机制，如下所示：

1. Spout的一个Task创建一个Tuple时，即在Spout的nextTuple()方法中实现从特定数据源读取数据的处理逻辑中，会与Acker进行通信，向Acker发送消息，Acker保存该Tuple对应信息：{spout-task task-id : val ack-val}}
2. Bolt在emit一个新的子Tuple时，会保存子Tuple与父Tuple的关系
3. 在Bolt中进行ack时，会计算出父Tuple与由该父Tuple新生成的所有子Tuple的一个异或值，将该值发送给Acker（计算异或值：tuple-id ^ (child-tuple-id1 ^ child-tuple-id2 … ^ child-tuple-idN)）。可见，这里Bolt并没有把所有生成的子Tuple发送给Acker，这要比发送一个异或值大得多了，只发送一个异或值大大降低了Bolt与Acker之间网络通信的开销
4. Acker收到Bolt发送的异或值，与当前保存的task-id对应的初始ack-val做异或，tuple-id与ack-val相同，异或结果为0，但是子Tuple的child-tuple-id等并不互相相同，只有等所有的子Tuple的child-tuple-id都执行ack回来，最后ack-val就为0，表示整个Tuple树处理成功。无论成功与失败，最后都要从Acker维护的队列中移除。
5. 最后，Acker会向产生该原始父Tuple的Spout对应的Task发送通知，成功或者失败，回调Spout的ack或fail方法。如果我们在实现Spout时，重写了ack和fail方法，处理回调就会执行这里的逻辑。

四. Storm设计：组件抽象

我们编写的处理业务逻辑的Topology提交到Storm集群后，就会发生任务的调度和资源的分配，从而也会基于Storm的设计，出现各种各样的组件。我们先看一下，Topology提交到Storm集群后的运行时部署分布图，如下图所示：



通过上图我们可以看出，一个Topology的Spout/Bolt对应的多个Task可能分布在多个Supervisor的多个Worker内部。而每个Worker内部又存在多个Executor，根据实际对Topology的配置在运行时进行计算分配。

从运行Topology的Supervisor节点，到最终的Task运行时对象，我们大概需要了解Storm抽象出来的一些概念，由于相对容易，我简单说明一下：

- **Topology**：Storm对一个分布式计算应用程序的抽象，目的是通过一个实现Topology能够完整地完成一件事情（从业务角度来看）。一个Topology是由一组静态程序组件（Spout/Bolt），组件关系Streaming Groups这两部分组成。
- **Spout**：描述了数据是如何从外部系统（或者组件内部直接产生）进入到Storm集群，并由该Spout所属的Topology来处理，通常是从一个数据源读取数据，也可以做一些简单的处理（为了不影响数据连续地、实时地、快速地进入到系统，通常不建议把复杂处理逻辑放在这里去做）。
- **Bolt**：描述了与业务相关的处理逻辑。

上面都是一些表达静态事物（组件）的概念，我们编写完成一个Topology之后，上面的组件都以静态的方式存在。下面，我们看一下提交Topology运行以后，会产生那些动态的组件（概念）：

- **Task**：Spout/Bolt在运行时所表现出来的实体，都称为Task，一个Spout/Bolt在运行时可能对应一个或多个Spout Task/Bolt Task，与实际在编写Topology时进行配置有关。
- **Worker**：运行时Task所在的一级容器，Executor运行于Worker中，一个Worker对应于Supervisor上创建的一个JVM实例
- **Executor**：运行时Task所在的直接容器，在Executor中执行Task的处理逻辑；一个或多个Executor实例可以运行在同一个Worker进程中，一个或多个Task可以运行于同一个Executor中；在Worker进程并行的基础上，Executor可以并行，进而Task也能够基于Executor实现并行计算