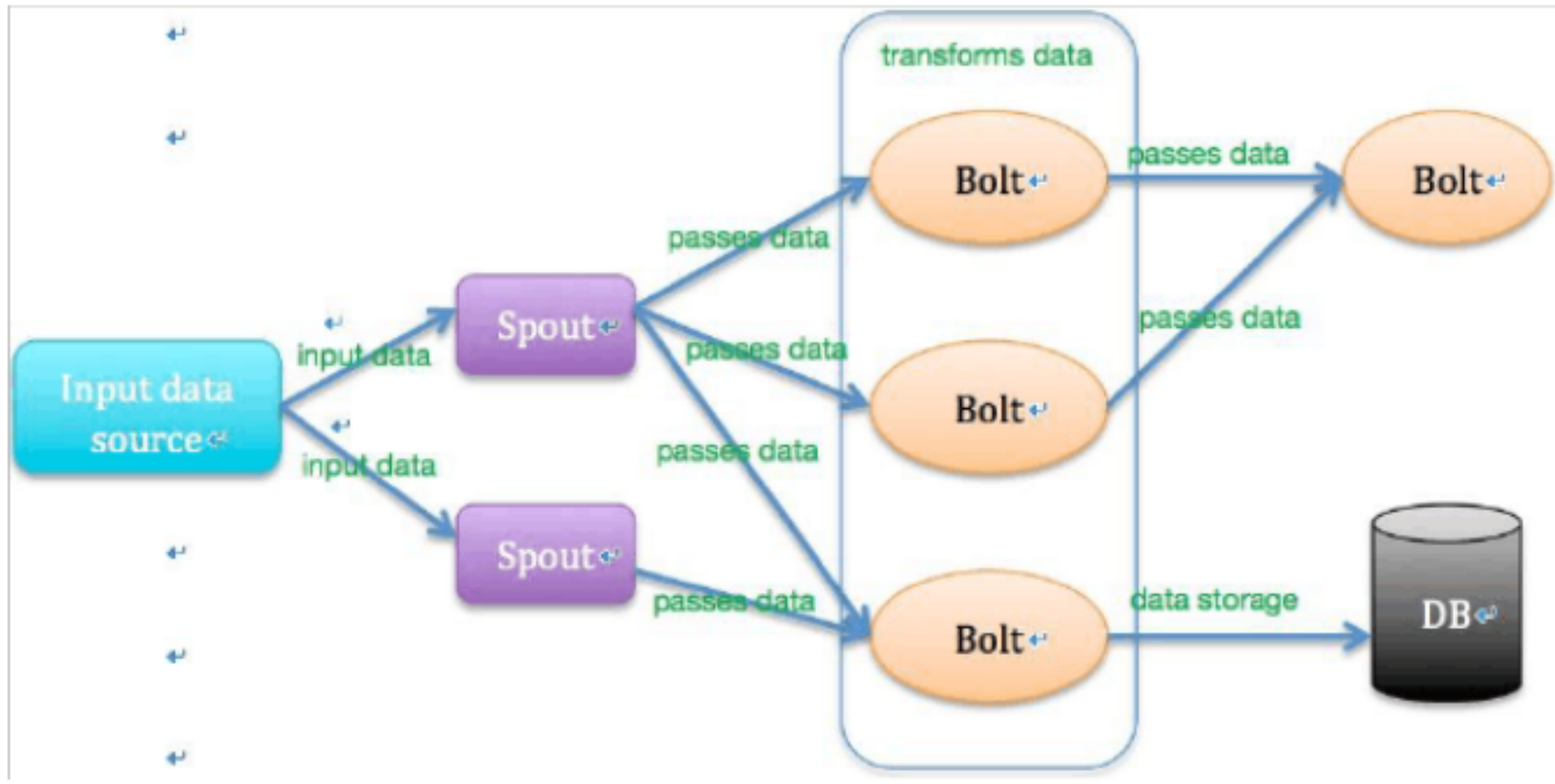
Storm 介绍

周龙鹏

一、数据处理过程



Storm 的术语解释

Storm 的术语包括 Stream、Spout、Bolt、Task、Worker、Stream Grouping 和 Topology。

Stream 是被处理的数据。

Spout 是数据源。

Bolt 处理数据。

Task 是运行于 Spout 或 Bolt 中的线程。

Worker 是运行这些线程的进程。

Stream Grouping 规定了 Bolt 接收什么东西作为输入数据。 数据可以随机分配（术语为 Shuffle），或者根据字段值分配（术语为 Fields），或者广播（术语为 All），或者总是发给一个 Task(术语为 Global)，也可以不关心该数据 （术语为 None），或者由自定义逻辑来决定（术语为 Direct）。Topology 是由 Stream Grouping 连接起来的 Spout 和 Bolt 节点网络。在 Storm Concepts 页面里对这些术语有更详细的描述。

（1） Topologies 用于封装一个实时计算应用程序的逻辑，类似于 Hadoop 的 MapReduce Job

（2）Stream 消息流， 是一个没有边界的 tuple 序列， 这些 tuples 会被以一种分布式的方式并行地创建和处理

（3）Spouts 消息源， 是消息生产者，他会从一个外部源读取数据并向 topology 里面面发出消息： tuple

（4）Bolts 消息处理者，所有的消息处理逻辑被封装在 bolts 里面，处理输入的数据流并产生输出的新数据流，可执行过滤，聚合，查询数据库等操作

（5）Task 每一个 Spout 和 Bolt 会被当作很多 task 在整个集群里面执行，每一个 task 对应到一个线程 .

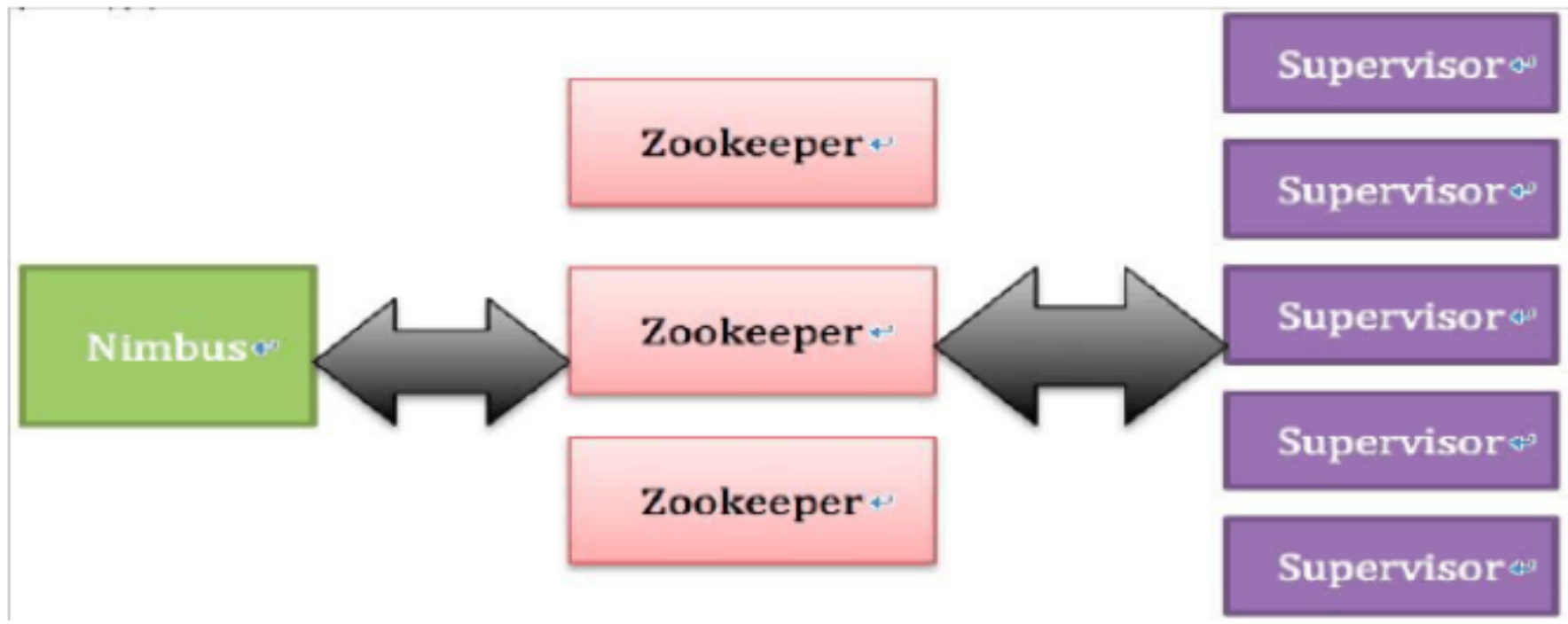
二、 storm 集群的组件（ topologies ）

Storm 集群非常类似 Hadoop 集群。Hadoop 上运行的是 MapReduce jobs，而 Storm 运行的是 topologies。Jobs和 topologies 本身是不同的，其中一个最大的不同就是， Mapreduce job 最终会结束，而 topology 则会持续的处理消息（直到你杀掉它）。

Storm 集群有 2 种节点： master （主节点）和 worker （工作节点）。

master 节点运行一个守护进程， 叫 Nimbus , 类似 Hadoop 中的 JobTracker。Nimbus 负责在集群中分发代码，分配任务，以及故障检测。

每个 worker 节点运行一个守护进程，叫 Supervisor 。Supervisor 监听分配到该服务器的任务，开始和结束工作进程。每个 worker 进程执行 topology 的一个子集；一个运行中的 topology 由许多分布在多台机器上的 worker 进程组成。



Nimbus 和 Supervisors 之间是通过 Zookeeper 协调。此外, Nimbus 和 Supervisor 是能快速失败（fail-fast ）和无状态的（stateless）;所有的状态都保存在 Zookeeper 或者在本地磁盘中。 这意味这你可以 kill －9 杀掉 Nimbus 或者 Supervisors，随后它们会自动恢复好像什么也没发生过。这项设计使得 Storm 集群变得非常稳定健壮。

三、 Topologies

在 Storm 中进行实时计算， 你可以创建所谓的 topologies 。一个 topology 是一个计算图（ a graph of computation ）。topology 的每个节点包括处理逻辑，节点之间数据如何传输的连接。

运行一个 topology 非常简单。将代码和相关依赖打包成一个简单 jar 包，运行如下命令：

```
storm jar all-my-code.jar backtype.storm.MyTopology arg1 arg2
```

将执行这个类： backtype.storm.MyTopology ， 参数 arg1 和 arg2。这个类的最主要功能定义了 topology , 并将其提交给 Nimbus 。storm jar 部分负责连接 Nimbus 和上传 jar。

因为 topology 定义是 Thrift 结构， Nimbus 是一个 Thrift service ， 所以你可以使用任何编程语言创建和提交 topologies 。上面的例子是最简单的方式，使用 JVM-based的语言。

四、 Streams

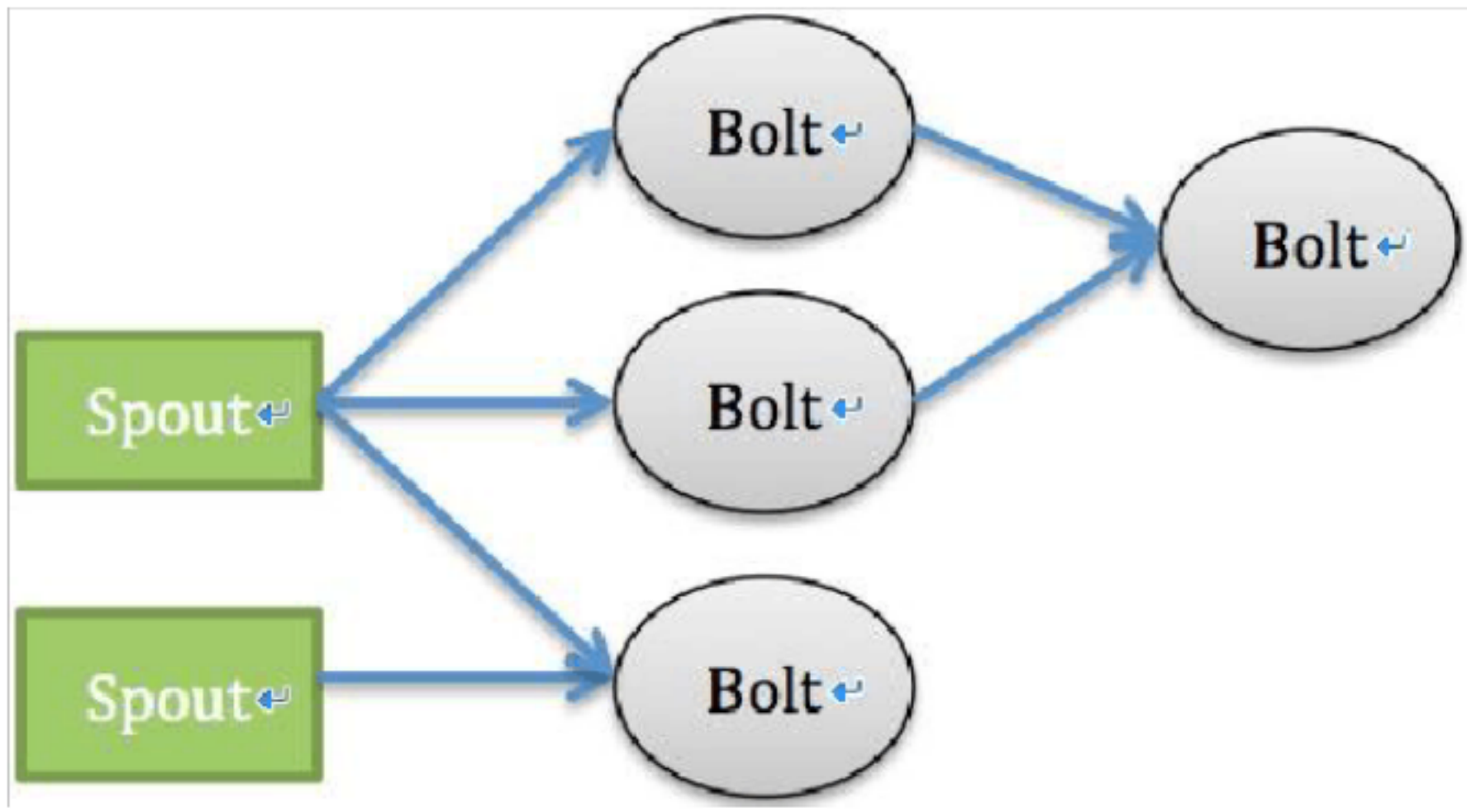
Storm 中最核心的抽象就是 stream。stream 是一个无边界的 tuples 序列。Storm 提供了基本流转换的分布式的可靠的方法。例如，你可以将 tweets 流转换成一个 trending topics 流。

基本的原始 Storm 提供了 spouts 和 bolts 的流转换。Spouts 和 bolts 提供了接口，实现你的应用逻辑。

spout 是流源头。例如，一个 spout 可以是从 Kestrel 队列中读取 tuples 并以流形式发射（ emit）出。或者一个 spout 可以连接 Twitter API，发出一个 tweets 流。

一个 bolt 使用任意数量的输入流，做些处理，可能再发射出新的流。复杂的流转换，例如从 tweets 流中计算出 trending topics 流，需要多个步骤，所以需要多个 bolts。bolts 可以做任何事情，包括运行函数、过滤 tuples、流的聚合、流连接、数据库交互等等。

spouts 和 bolts 的网络封装到 topology，后者是你提交给 storm 集群运行的最上层的抽象。一个 topology 是一个由 spout 或 bolt 节点组成的流变换的图，图的边指示哪个 bolts 订阅给哪个 stream。当一个 spout 或 bolt 发射一个 tuple 给 stream，它将发送这个 tuple 到订阅了这个 stream 的每一个 bolt。



在 topology 中，节点之间的连接指示这 tuples 如何传递。例如，如果 Spout A 和 Bolt B 之间存在连接， Spout A 和 Bolt C 之间存在连接， Bolt B 和 Bolt C 之间存在连接，那么每时刻 Spout A 发送出一个 tuple，它将发给 Bolt B 和 Bolt C，所有 Bolt B 的输出 tuple 将都会发给 Bolt C。

每个 Storm topology 中，在各个节点中都是并行执行。在 topology 中，你可以指定每个节点的并行数，然后 Storm 将会开启相应数量的线程来运行。

每个 topology 会永久运行，直到你 kill 它。Storm 会自动再分配失败的任务。此外， Storm 会保证不会有数据丢失，即使是宕机消息丢失。

五、 Data Model

Storm 使用 tuples 作为它的数据模型。 每个 tuple 是值的名字序列 （a named list of values），且 tuple 中域（ field ）可以是任何类型对象。 Storm 支持所有的原始类型， strings 、byte arrays，作为 tuple field values。如果要使用新类型对象，需要为这个类型实现一个序列化（ a serializer ）。

每个节点都必须声明它发送出的 tuple 的输出域。 例如， 这个 bolt 声明了它发送 2 个 tuple ， 对应的域类型是 double 和 triple

```
public class DoubleAndTripleBolt extends BaseRichBolt {
    private 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"));
    }
}
```

The `declareOutputFields` function declares the output fields `["double", "triple"]` for the component. The rest of the bolt will be explained in the upcoming sections.

六、 A simple topology

来看看一个简单 topology 实例， 深入探索相关的概念和编码。下面是 ExclamationTopology 的定义：

```
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("words", new TestWordSpout(), 10);
builder.setBolt("exclaim1", new ExclamationBolt(), 3)
        .shuffleGrouping("words");
builder.setBolt("exclaim2", new ExclamationBolt(), 2)
        .shuffleGrouping("exclaim1");
```

这个 topology 包括 1 个 spout 和 2 个 bolts。这个 spout 发送 words， 每个 bolt 将输入字符串附加上“ ！ ！!”。各节点安排在一条线： spout 发给第一个 bolt，后者在发给第二个 bolt。例如 spout 发送 [“ bob ”] and [“ john ”]，经过两个 bolt，将会发送出 [“ bob!!!!!! ”] and [“ john!!!!!! ”]。

This code defines the nodes using the `setSpout` and `setBolt` methods. These methods take as input a user-specified id, an object containing the processing logic, and the amount

of parallelism you want for the node. In this example, the spout is given id "words" and the bolts are given ids "exclaim1" and "exclaim2".

这个对象包含两个接口的处理逻辑的实现： spout 的一个接口 IRichSpout， bolt 的一个接口 IRichBolt。

最后一个参数， 你希望节点上并行数， 是可选项。 它声明了这个集群中由多少个线程来运行这个组件。 如果省略这个参数， Storm 缺省下只分配一个线程给一个节点。

setBolt 返回一个 InputDeclarer 对象，用于声明 Bolt 的输入。这个例子中，“ exclaim1 ”定义了它需要读取所有来自 “ words ”组件以 shuffle grouping 方式发出的 tuples，“ exclaim2 ”定义了需要读取所有来自 “ exclaim1 ”组件以 shuffle grouping 方式发出的 tuples。shuffle grouping 意思是将 tuples 从输入任务中随机分配给 bolts 任务。关于组件之间分组数据的方式将后面 grouping 章节。

如果你想让组件 exclaim2 同时读取 words 组件和 exclaim1 组件两者的 tuples，你可以这个定义 exclaim2，输入定义可以指定多个来源，形成链式：

```
builder.setBolt("exclaim2", new ExclamationBolt(), 5)
            .shuffleGrouping("words")
            .shuffleGrouping("exclaim1");
```

As you can see, input declarations can be chained to specify multiple sources for the Bolt.

进一步看看这个 topology 中 spout 和 bolt 的实现。 Spouts 负责产生新的消息给这个 topology。 TestWordSpout 是从一个列表中每隔 100ms 随机选择单词生成一个 tuple。在 TestWordSpout 的 nextTuple() 实现如下：

```
public void nextTuple() {
    Utils.sleep(100);
    final String[] words = new String[] {"nathan", "mike", "jackson", "golda", "bertels"};
    final Random rand = new Random();
    final String word = words[rand.nextInt(words.length)];
    _collector.emit(new Values(word));
}
```

As you can see, the implementation is very straightforward.

ExclamationBolt appends the string "!!!" to its input. Let's take a look at the full implementation for ExclamationBolt:

```
public static class ExclamationBolt implements IRichBolt {
    OutputCollector _collector;

    @Override
    public void prepare(Map conf, TopologyContext context, OutputCollector collector) {
        _collector = collector;
    }

    @Override
    public void execute(Tuple tuple) {
        _collector.emit(tuple, new Values(tuple.getString(0) + "!!!"));
        _collector.ack(tuple);
    }

    @Override
    public void cleanup() {
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word"));
    }

    @Override
    public Map getComponentConfiguration() {
        return null;
    }
}
```

其中 prepare 方法提供给 bolt 一个 OutputCollector ,用于发送出一个 tuple 。Tuple 可以被任何时间发送，在 prepare 、execute、cleanup 方法，甚至是其他线程中异步方式。这里的 prepare 实现的很简单，将 OutputCollector 存成一个实例变量，在后面的 execute 方法中使用。

其中 execute 方法是从一个 bolt 输入中接收一个 tuple ， ExclaimationBolt 提取 tuple 中第一个域，附加上“ ！ !!”成一个新字符串，发送成一个新的 tuple 。如果你实现一个 bolt 读取多个输入源，你可以使用 Tuple#getSourceComponent 方法找到 Tuple 来自哪个组件。 execute 中执行了一些方法：输入 tuple 被作为 emit 的第一个参数，输入 tuple is acked on the final line 。这些 API 是 Storm 保证不丢失数据的基础。

其中 cleanup 方法，是在 Bolt 关闭时调用，需要清理掉所有被打开的资源。在集群中并不会保证执行这个方法： 例如，服务器宕机了， 是没有办法调用这个方法的。 cleanup 方法主要还是用于你以 local mode 方式运行 topologies ，在运行和和杀掉很多 topologies 时没有资源泄露。

其中 declareOutputFields 方法声明了 ExclamationBolt 发送的 1-tuples ，是以名叫 word 的一个域。
其中 getComponentConfiguration 方法允许你为这个组件运行配置不同参数。参看 Configuration 说明。
方法 cleanup 和 getComponentConfiguration 通常在 bolt 实现时是不需要的。你可以通过继承 BaseRichBolt（这个基类提供了一个缺省的实现） 来更简单的定义一个 bolt。如下：

```java
public static class ExclamationBolt extends BaseRichBolt {
    OutputCollector _collector;

    @Override
    public void prepare(Map conf, TopologyContext context, OutputCollector collector) {
        _collector = collector;
    }

    @Override
    public void execute(Tuple tuple) {
        _collector.emit(tuple, new Values(tuple.getString(0) + "!!!"));
        _collector.ack(tuple);
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word"));
    }
}
```

七、 Running ExclamationTopology in local mode

Storm 有两种操作模式：local mode 和 distributed mode 。在 local mode 中, Storm 通过线程来模拟 worker nodes，全部运行在进程中。 Local mode 主要是方便 topologies 的测试和开发。在 distributed mode 中，提交 topology 给 master 同时，也需要将运行的代码提交。 master 会分发代码，并分配 worker 来运行 topology 。如果 worker 挂了， master 会重新分配任务到其他地方。

下面是以 local mode 方式运行 ExclamationTopology 的代码：

```java
Config conf = new Config();
conf.setDebug(true);
conf.setNumWorkers(2);

LocalCluster cluster = new LocalCluster();
cluster.submitTopology("test", conf, builder.createTopology());
Utils.sleep(10000);
cluster.killTopology("test");
cluster.shutdown();
```

First, the code defines an in-process cluster by creating a `LocalCluster` object. Submitting topologies to this virtual cluster is identical to submitting topologies to distributed clusters. It submits a topology to the `LocalCluster` by calling `submitTopology`, which takes as arguments a name for the running topology, a configuration for the topology, and then the topology itself.

The name is used to identify the topology so that you can kill it later on. A topology will run indefinitely until you kill it.

The configuration is used to tune various aspects of the running topology. The two configurations specified here are very common:

1. TOPOLOGY_WORKERS (set with `setNumWorkers`) specifies how many processes you want allocated around the cluster to execute the topology. Each component in the topology will execute as many threads. The number of threads allocated to a given component is configured through

the `setBolt` and `setSpout` methods. Those threads exist within worker processes. Each worker process contains within it some number of threads for some number of components. For instance, you may have 300 threads specified across all your components and 50 worker processes specified in your config. Each worker process will execute 6 threads, each of which of could belong to a different component. You tune the performance of Storm topologies by tweaking the parallelism for each component and the number of worker processes those threads should run within.

2. TOPOLOGY_DEBUG (set with `setDebug`), when set to true, tells Storm to log every message every emitted by a component. This is useful in local mode when testing topologies, but you probably want to keep this turned off when running topologies on the cluster.

There's many other configurations you can set for the topology. The various configurations are detailed on the Javadoc for Config.

To learn about how to set up your development environment so that you can run topologies in local mode (such as in Eclipse), see Creating a new Storm project.

八、 Stream grouping 分类

1、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 数目相同.

2、Fields Grouping :按字段分组，比如按 userid 来分组，具有同样 userid 的 tuple 会被分到相同的 Bolts，而不同的 userid 则会被分配到不同的 Bolts.

3、All Grouping ：广播发送，对于每一个 tuple，所有的 Bolts 都会收到.

4、Global Grouping: 全局分组，这个 tuple 被分配到 storm 中的一个 bolt 的其中一个 task.再具体一点就是分配给 id 值最低的那个 task.

5、Non Grouping: 不分组，意思是说 stream 不关心到底谁会收到它的 tuple.目前他和 Shuffle grouping 是一样的效果,有点不同的是 storm 会把这个 bolt 放到这个 bolt 的订阅者同一个线程去执行.

6、Direct Grouping: 直接分组,这是一种比较特别的分组方法，用这种分组意味着消息的发送者举鼎由消息接收者的哪个 task 处理这个消息.只有被声明为 Direct Stream 的消息流可以声明这种分组方法.而且这种消息 tuple 必须使用 emitDirect 方法来发射.消息处理者可以通过 TopologyContext 来或者处理它的消息的 taskid (OutputCollector.emit 方法也会返回 taskid)

九、 Storm 如何保证消息被处理

storm 保证每个 tuple 会被 topology 完整的执行。storm 会追踪由每个 spout tuple 所产生的 tuple 树（一个 bolt 处理一个 tuple 之后可能会发射别的 tuple 从而可以形成树状结构），并且跟踪这棵 tuple 树什么时候成功处理完。每个 topology 都有一个消息超时的设置，如果 storm 在这个超时的时间内检测不到某个 tuple 树到底有没有执行成功， 那么 topology 会把这个 tuple 标记为执行失败， 并且过

一会会重新发射这个 tuple。一个 tuple 能根据新获取到的 spout 而触发创建基于此的上千个 tuple