Storm 基础知识

一、概述

Storm 是一个开源的分布式实时计算系统，可以简单、可靠的处理大量的数据流。Storm 有很多使用场景：如实时分析，在线机器学习，持续计算，分布式 RPC，ETL 等等。Storm 支持水平扩展，具有高容错性，保证每个消息都会得到处理，而且处理速度很快（在一个小集群中，每个结点每秒可以处理数以百万计的消息）。Storm 的部署和运维都很便捷，而且更为重要的是可以使用任意编程语言来开发应用。

1、Storm 特点

1、编程模型简单

在大数据处理方面相信大家对 hadoop 已经耳熟能详，基于 Google Map/Reduce 来实现的 Hadoop 为开发者提供了 map、reduce 原语，使并行批处理程序变得非常地简单和优美。同样，Storm 也为大数据的实时计算提供了一些简单优美的原语，这大大降低了开发并行实时处理的任务的复杂性，帮助你快速、高效的开发应用。

2、可扩展

在 Storm 集群中真正运行 topology 的主要有三个实体：工作进程（workers）、线程(executor)和任务(task)。Storm 集群中的每台机器上都可以运行多个工作进程，每个工作进程又可创建多个线程，每

个线程可以执行多个任务，任务是真正进行数据处理的实体，我们开发的 spout、bolt 就是作为一个或者多个任务的方式执行的。

因此，计算任务在多个线程、进程和服务器之间并行进行，支持灵活的水平扩展。

3、高可靠性

Storm 可以保证 spout 发出的每条消息都能被“完全处理”，这也是直接区别于其他实时系统的地方，如 S4。

请注意，spout 发出的消息后续可能会触发产生成千上万条消息，可以形象的理解为一棵消息树，其中 spout 发出的消息为树根，Storm 会跟踪这棵消息树的处理情况，只有当这棵消息树中的所有消息都被处理了，Storm 才会认为 spout 发出的这个消息已经被“完全处理”。如果这棵消息树中的任何一个消息处理失败了，或者整棵消息树在限定的时间内没有“完全处理”，那么 spout 发出的消息就会重发。

考虑到尽可能减少对内存的消耗，Storm 并不会跟踪消息树中的每个消息，而是采用了一些特殊的策略，它把消息树当作一个整体来跟踪，对消息树中所有消息的唯一 id 进行异或计算，通过是否为零来判定 spout 发出的消息是否被“完全处理”，这极大的节约了内存和简化了判定逻辑，后面会对这种机制进行详细介绍。

这种模式，每发送一个消息，都会同步发送一个ack/fail，对于网络的带宽会有一定的消耗，如果对于可靠性要求不高，可通过使用不同的emit接口关闭该模式。

上面所说的，Storm保证了每个消息至少被处理一次，但是对于有些计算场合，会严格要求每个消息只被处理一次，幸而Storm的0.7.0引入了事务性拓扑，解决了这个问题，后面会有详述。

4、高容错性

如果在消息处理过程中出了一些异常，Storm会重新安排这个出问题的topology。Storm保证一个topology永远运行（除非你显式杀掉这个topology）。

当然，如果topology中存储了中间状态，那么当topology重新被Storm启动的时候，需要应用自己处理中间状态的恢复。

5、支持多种编程语言

除了用java实现spout和bolt，你还可以使用任何你熟悉的编程语言来完成这项工作，这一切得益于Storm所谓的多语言协议。多语言协议是Storm内部的一种特殊协议，允许spout或者bolt使用标准输入和标准输出来进行消息传递，传递的消息为单行文本或者是json编码的多行。

Storm支持多语言编程主要是通过ShellBolt，ShellSpout和ShellProcess这些类来实现的，这些类都实现了IBolt 和 ISpout

接口，以及让 shell 通过 java 的 ProcessBuilder 类来执行脚本或者程序的协议。

可以看到，采用这种方式，每个 tuple 在处理的时候都需要进行 json 的编解码，因此在吞吐量上会有较大影响。

6、支持本地模式

Storm 有一种“本地模式”，也就是在进程中模拟一个 Storm 集群的所有功能，以本地模式运行 topology 跟在集群上运行 topology 类似，这对于我们开发和测试来说非常有用。

7、高效

用 ZeroMQ 作为底层消息队列，保证消息能快速被处理。

二、Storm 基本概念

在运行一个 Storm 任务之前，需要了解一些概念：

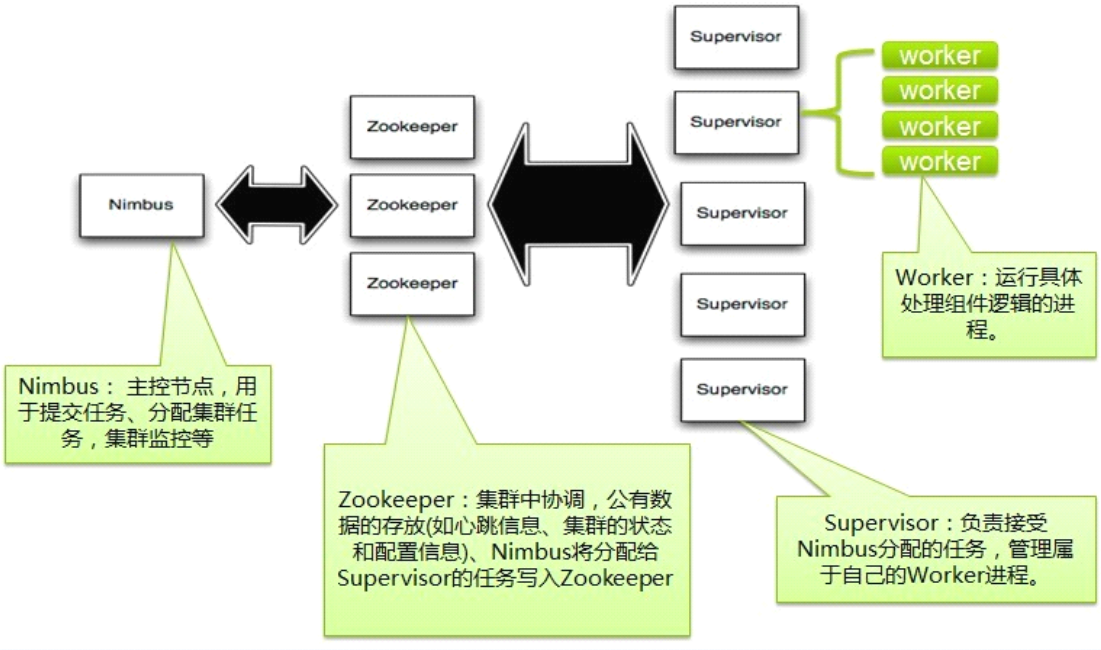
- ✓ Topologies
- ✓ Streams
- ✓ Spouts
- ✓ Bolts
- ✓ Stream groupings
- ✓ Reliability
- ✓ Tasks

✓ Workers

✓ Configuration

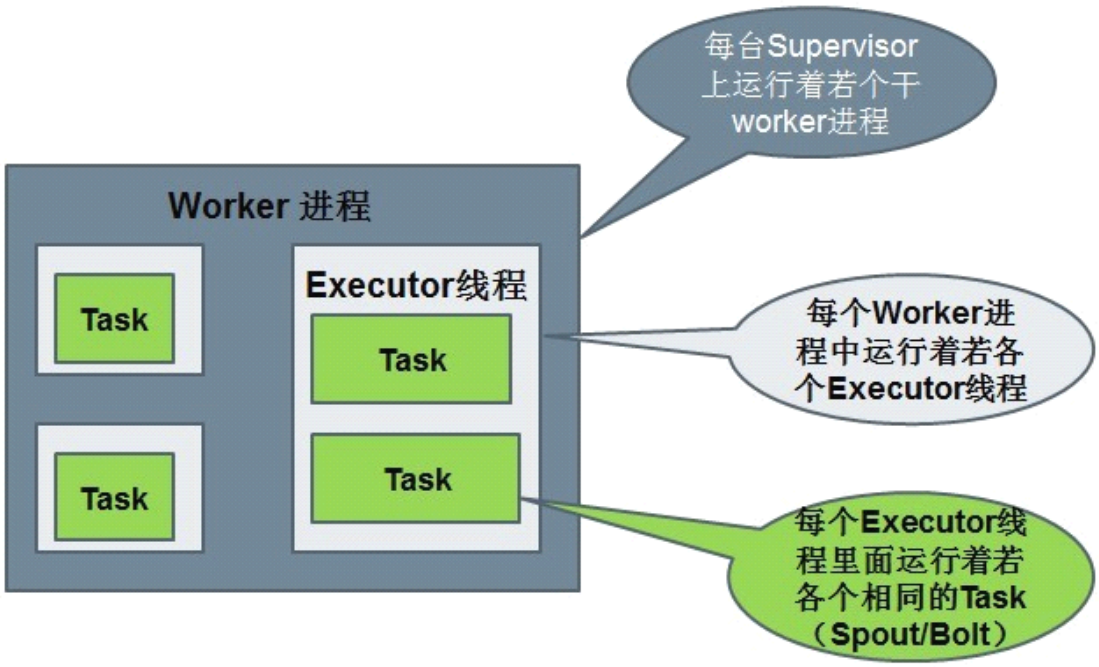
Storm 集群和 Hadoop 集群表面上看很类似。但是 Hadoop 上运行的是 MapReduce jobs，而在 Storm 上运行的是拓扑（topology），这两者之间是非常不一样的。一个关键的区别是： 一个 MapReduce job 最终会结束， 而一个 topology 永远会运行（除非你手动 kill 掉）。

在 Storm 的集群里面有两种节点： 控制节点（master node）和工作节点（worker node）。控制节点上面运行一个叫 Nimbus 后台程序，它的作用类似 Hadoop 里面的 JobTracker。Nimbus 负责在集群里面分发代码，分配计算任务给机器， 并且监控状态。



每一个工作节点上面运行一个叫做 Supervisor 的节点。Supervisor 会监听分配给它那台机器的工作，根据需要启动/关闭 worker 进程。每个 supervisor 上运行着若干个 worker 进程（根据

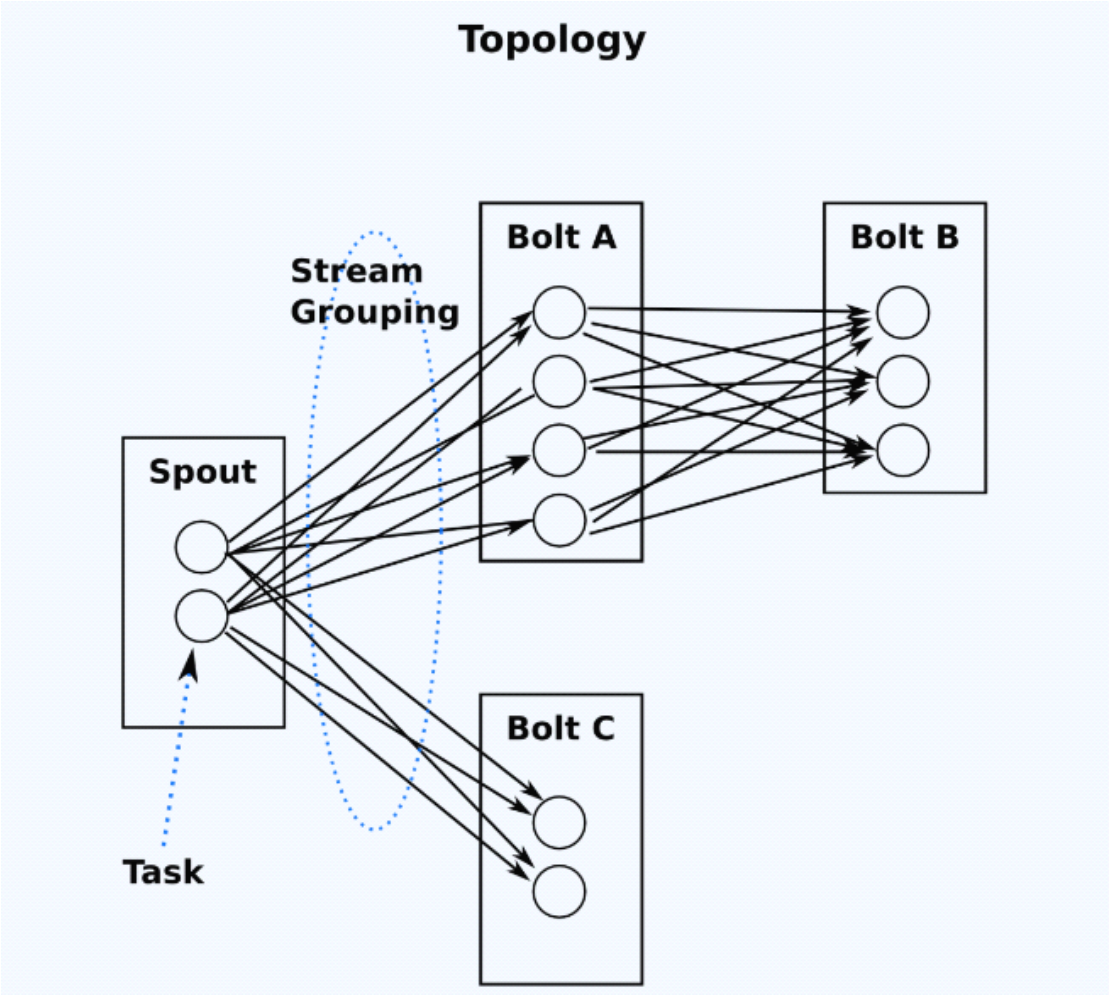
配置文件 supervisor.slots.ports 进行配置)。 在一个 worker 进程中，又包含多个 Executor 线程，一个 Executor 线程中，可以包含一个或多个相同的 Task（spout/bolt），默认一个 Executor 线程包含一个 task。



Nimbus 和 Supervisor 之间的所有协调工作都是通过 Zookeeper 集群完成。另外，Nimbus 进程和 Supervisor 进程都是快速失败（fail-fast)和无状态的。所有的状态要么在 zookeeper 里面， 要么在本地磁盘上。这也就意味着你可以用 kill -9来杀死 Nimbus 和 Supervisor 进程， 然后再重启它们，就好像什么都没有发生过。这个设计使得 Storm 异常的稳定。

1、Topologies

一个 topology 是 spouts 和 bolts 组成的图， 通过 stream groupings 将图中的 spouts 和 bolts 连接起来，如下图：



一个 topology 会一直运行直到你手动 kill 掉，Storm 自动重新分配执行失败的任务， 并且 Storm 可以保证你不会有数据丢失（如果开启了高可靠性的话）。如果一些机器意外停机它上面的所有任务会被转移到其他机器上。

运行一个 topology 很简单。首先，把你所有的代码以及所依赖的 jar 打进一个 jar 包。然后运行类似下面的这个命令：

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

这个命令会运行主类：backtype.strom.MyTopology，参数是arg1, arg2。这个类的main函数定义这个topology并且把它提交给Nimbus。storm jar负责连接到Nimbus并且上传jar包。

Topology的定义是一个Thrift结构，并且Nimbus就是一个Thrift服务，你可以提交由任何语言创建的topology。上面的方面是用JVM-based语言提交的最简单的方法。

如下代码即定义了一个topology：

```
TopologyBuilder builder = new TopologyBuilder();

builder.setSpout("spout", new RandomSentenceSpout(), 5);

builder.setBolt("split", new SplitSentence(), 8).shuffleGrouping("spout");

builder.setBolt("count", new WordCount(), 12).fieldsGrouping("split", new Fields("word"));
```

2、Streams

消息流stream是storm里的关键抽象。一个消息流是一个没有边界的tuple序列，而这些tuple序列会以一种分布式的方式并行地创建和处理。通过对stream中tuple序列中每个字段命名来定义stream。在默认的情况下，tuple的字段类型可以是：integer，long，short，byte，string，double，float，boolean和byte array。你也可以自定义类型（只要实现相应的序列化器）。

```java
/**
 * 信息流定义方法
 */
public void declareOutputFields(OutputFieldsDeclarer declarer) {
    //默认ID的信息流定义
    declarer.declare(new Fields("word", "count"));
    //自定义ID的消息流定义
    declarer.declareStream("streamId", new Fields("word", "count"));
}
```

每个消息流在定义的时候会被分配给一个 id，因为单向消息流使用的相当普遍， OutputFieldsDeclarer 定义了一些方法让你可以定义一个 stream 而不用指定这个 id。在这种情况下这个 stream 会分配个值为‘default’默认的 id 。

Storm 提供的最基本的处理 stream 的原语是 spout 和 bolt。你可以实现 spout 和 bolt 提供的接口来处理你的业务逻辑。

3、Spouts

消息源 spout 是 Storm 里面一个 topology 里面的消息生产者。一般来说消息源会从一个外部源读取数据并且向 topology 里面发出消息：tuple。Spout 可以是可靠的也可以是不可靠的。如果这个 tuple 没有被 storm 成功处理，可靠的消息源 spouts 可以重新发射一个 tuple， 但是不可靠的消息源 spouts 一旦发出一个 tuple 就不能重发了。

消息源可以发射多条消息流 stream 。使用 OutputFieldsDeclarer.declareStream 来定义多个 stream，然后使用 SpoutOutputCollector 来发射指定的 stream。

```java
/**
 * 定义了2个消息流
 */
public void declareOutputFields(OutputFieldsDeclarer declarer) {
    declarer.declareStream("streamId1", new Fields("word1"));
    declarer.declareStream("streamId2", new Fields("word2"));
}
/**
 * 根据消息流发射相应的消息
 */
public void nextTuple() {
  collector.emit("streamId1", new Values("streamid1's word1"));
  collector.emit("streamId2", new Values("streamid2's word2"));
}
```

Spout 类里面最重要的方法是 nextTuple。要么发射一个新的 tuple 到 topology 里面或者简单的返回如果已经没有新的 tuple。要注意的是 nextTuple 方法不能阻塞，因为 storm 在同一个线程上面调用所有消息源 spout 的方法。

另外两个比较重要的 spout 方法是 ack 和 fail。storm 在检测到一个 tuple 被整个 topology 成功处理的时候调用 ack，否则调用 fail。storm 只对可靠的 spout 调用 ack 和 fail。

4、Bolts

所有的消息处理逻辑被封装在 bolts 里面。Bolts 可以做很多事情：过滤，聚合，查询数据库等等。

Bolts 可以简单的做消息流的传递。复杂的消息流处理往往需要很多步骤，从而也就需要经过很多 bolts。

Bolts 和 spouts 一样，可以发射多条消息流， 使用 OutputFieldsDeclarer.declareStream 定义 stream ， 使用 OutputCollector.emit 来选择要发射的 stream。

Bolts 的主要方法是 execute, 它以一个 tuple 作为输入，使用 OutputCollector 来发射 tuple，bolts 必须要为它处理的每一个 tuple 调用 OutputCollector 的 ack 方法，以通知 Storm 这个 tuple 被处理完成了，从而通知这个 tuple 的发射者 spouts。一般的流程是： bolts 处理一个输入 tuple， 发射0个或者多个 tuple, 然后调用 ack 通知 storm 自己已经处理过这个 tuple 了。storm 提供了一个 IBasicBolt 会自动调用 ack。

5、Stream groupings

定义一个 topology 的其中一步是定义每个 bolt 接收什么样的流作为输入。stream grouping 就是用来定义一个 stream 应该如果分配数据给 bolts 上面的多个 tasks。

Storm 里面有7种类型的 stream grouping：

- ✓ 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。
- ✓ 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 里的一个 task， 而不同的 userid 则会被分配到不同的 bolts 里的 task。

- ✓ All Grouping：广播发送，对于每一个 tuple，所有的 bolts 都会收到。
- ✓ 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。
- ✓ Non Grouping：不分组，这个分组的意思是说 stream 不关心到底谁会收到它的 tuple。目前这种分组和 Shuffle grouping 是一样的效果， 有一点不同的是 storm 会把这个 bolt 放到这个 bolt 的订阅者同一个线程里面去执行。
- ✓ 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指定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 来获取处理它的消息的 task 的 id （OutputCollector.emit 方法也会返回 task 的 id）。
- ✓ Local or shuffle grouping：如果目标 bolt 有一个或者多个 task 在同一个工作进程中，tuple 将会被随机发生给这些 tasks。否则，和普通的 Shuffle Grouping 行为一致。

代码示例：

```
TopologyBuilder builder = new TopologyBuilder();

builder.setSpout("spout", new RandomSentenceSpout(), 5);
builder.setBolt("split", new SplitSentence(), 8).shuffleGrouping("spout");
builder.setBolt("count", new WordCount(), 12).fieldsGrouping("split", new Fields("word"));
```

6、Reliability

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，并且过一会儿重新发射这个 tuple。

为了利用 Storm 的可靠性特性，在你发出一个新的 tuple 以及你完成处理一个 tuple 的时候你必须要通知 storm。这一切是由 OutputCollector 来完成的。通过 emit 方法来通知一个新的 tuple 产生了，通过 ack 方法通知一个 tuple 处理完成了。

Storm 的可靠性我们在第四章会深入介绍。

7、Tasks

每一个 spout 和 bolt 会被当作很多 task 在整个集群里执行。每一个 executor 对应到一个线程，在这个线程上运行多个 task，而 stream grouping 则是定义怎么从一堆 task 发射 tuple 到另外一堆 task。你可以调用 TopologyBuilder 类的 setSpout 和 setBolt 来设置并行度（也就是有多少个 task)。

代码示例如下：

```
TopologyBuilder builder = new TopologyBuilder();

builder.setSpout("spout", new RandomSentenceSpout(), 5).setNumTasks(10);
builder.setBolt("split", new SplitSentence(), 8).shuffleGrouping("spout").setNumTasks(8);
builder.setBolt("count", new WordCount(), 12).fieldsGrouping("split", new Fields("word")).setNumTasks(24);
```

表示"spout"的线程数为5，任务数为10，即一个线程里运行2个任务；"split"则为一个线程运行一个任务，和默认的一致。

8、Workers

一个 topology 可能会在一个或者多个 worker（工作进程）里面执行，每个 worker 是一个物理 JVM 并且执行整个 topology 的一部分。比如，对于并行度是300的 topology 来说，如果我们使用50个工作进程来执行，那么每个工作进程会开启6个线程，默认每个线程处理一个 tasks。Storm 会尽量均匀的工作分配给所有的 worker。每个 supervisor 上运行着若干个 worker 进程（根据配置文件 supervisor.slots.ports 进行配置）。

9、Configuration

Storm 里面有一堆参数可以配置来调整 Nimbus, Supervisor 以及正在运行的 topology 的行为，一些配置是系统级别的，一些配置是 topology 级别的。default.yaml 里面有所有的默认配置。你可以通过定义个 storm.yaml 在你的 classpath 里来覆盖这些默认配置。并且你也可以在代码里面设置一些 topology 相关的配置信息（使用 StormSubmitter）。

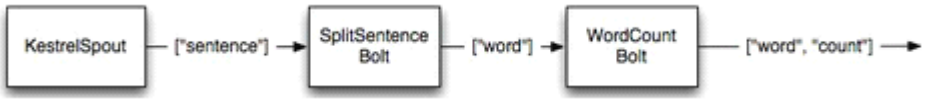
三、构建 Topology

1、实现的目标：

我们将设计一个 topology，来实现对一个句子里面的单词出现的频率进行统计。这是一个简单的例子，目的是让大家对于 topology 快速上手，有一个初步的理解。

2、设计 Topology 结构：

在开始开发 Storm 项目的第一步，就是要设计 topology。确定好你的数据处理逻辑，我们今天将的这个简单的例子， topology 也非常简单。整个 topology 如下：

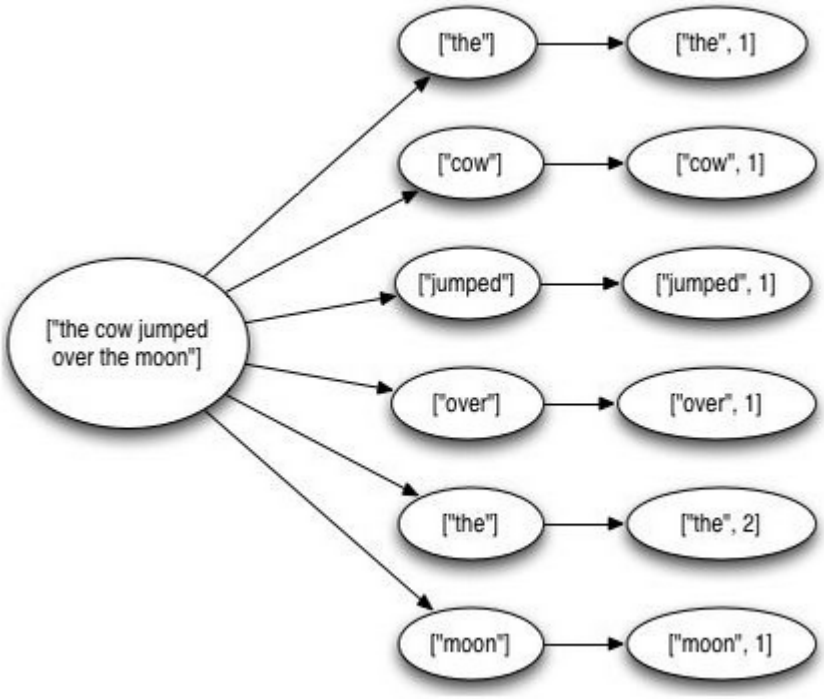


整个 topology 分为三个部分：

- ✓ KestrelSpout:数据源，负责发送 sentence
- ✓ Splitsentence:负责将 sentence 切分
- ✓ Wordcount:负责对单词的频率进行累加

3、设计数据流

这个 topology 从 kestrel queue 读取句子,并把句子划分成单词,然后汇总每个单词出现的次数,一个 tuple 负责读取句子,每一个 tuple 分别对应计算每一个单词出现的次数,大概样子如下所示：



4、代码实现：

- 构建 maven 环境

使用 maven 的话，把下面的配置添加在你项目的 pom.xml 里面。

```
<!--repository 最好直接写在 maven 的 setting 文件中-->
<repository>
    <id>clojars.org</id>
    <url>http://clojars.org/repo</url>
</repository>
<dependency>
          <groupId>org.apache.curator</groupId>
          <artifactId>curator-recipes</artifactId>
          <version>2.4.0</version>
      </dependency>
      <dependency>
          <groupId>org.apache.storm</groupId>
          <artifactId>storm-core</artifactId>
          <version>0.9.2-incubating</version>
          <scope>provided</scope>
 </dependency>
```

- 定义 topology：

```
public static void main(String[] args) throws Exception {

  TopologyBuilder builder = new TopologyBuilder();

  builder.setSpout("spout", new RandomSentenceSpout(), 5);

  builder.setBolt("split", new SplitSentence(), 8).shuffleGrouping("spout");
  builder.setBolt("count", new WordCount(), 12).fieldsGrouping("split", new Fields("word"));

  Config conf = new Config();
  conf.setDebug(true);


  if (args != null && args.length > 0) {
    conf.setNumWorkers(3);

    StormSubmitter.submitTopology(args[0], conf, builder.createTopology());
  }
  else {
    conf.setMaxTaskParallelism(3);

    LocalCluster cluster = new LocalCluster();
    cluster.submitTopology("word-count", conf, builder.createTopology());

    Thread.sleep(10000);

    cluster.shutdown();
  }
}
```

这个 topology 的 spout 从句子队列中读取句子。

Spout 用 setSpout 方法插入一个独特的 id 到 topology。Topology 中的每个节点必须给予一个 id，id 是由其他 bolts 用于订阅该节点的输出流。

setBolt 是用于在 Topology 中插入 bolts。 在 topology 中定义的第一个 bolts 是切割句子的 bolts。 这个 bolts 将句子流转成成单词流。具体代码见 storm-starter。

5、运行 Topology

storm 的运行有两种模式：本地模式和分布式模式.

- 本地模式：

storm 用一个进程里面的线程来模拟所有的 spout 和 bolt. 本地模式对开发和测试来说比较有用。 你运行 storm-starter 里面的 topology 的时候它们就是以本地模式运行的，你可以看到 topology 里面的每一个组件在发射什么消息。

下面是以本地模式运行的代码：

```
Config conf = new Config();
conf.setDebug(true);

conf.setMaxTaskParallelism(3);

LocalCluster cluster = new LocalCluster();
cluster.submitTopology("word-count", conf, builder.createTopology());

Thread.sleep(10000);

cluster.shutdown();
```

首先， 这个代码定义通过定义一个 LocalCluster 对象来定义一个进程内的集群。提交 topology 给这个虚拟的集群和提交 topology 给分布式集群是一样的。通过调用 submitTopology 方法来提交 topology， 它接受三个参数：要运行的 topology 的名字，一个配置对象以及要运行的 topology 本身。

topology 的名字是用来唯一区别一个 topology 的，这样你然后可以用这个名字来杀死这个 topology 的。前面已经说过了， 你必须显式的杀掉一个 topology， 否则它会一直运行。

- 分布式模式：

storm 由一堆机器组成。当你提交 topology 给 master（nimbus）的时候， 你同时也把 topology 的代码提交了。master 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了， master 节点会把认为重新分配到其它节点。

6、Config 配置

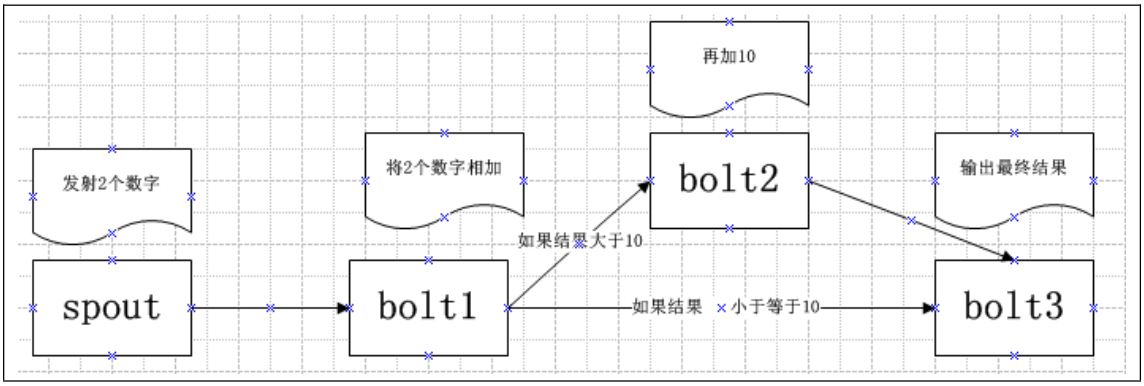
Conf 对象可以配置很多东西， 下面两个是最常见的：

TOPOLOGY_WORKERS(setNumWorkers) 定义你希望集群分配多少个工作进程给你来执行这个 topology. 你可以通过调整每个组件的并行度以及这些线程所在的进程数量来调整 topology 的性能。

TOPOLOGY_DEBUG(setDebug)， 当它被设置成 true 的话， storm 会记录下每个组件所发射的每条消息。这在本地环境调试 topology 很有用， 但是在线上这么做的话会影响性能的。

7、一个有分支的 topology 示例

Topology 图如下：



Topology 代码定义如下：

```
public class BranchTotoplogy {

    public static void main(String[] args) {
        TopologyBuilder builder = new TopologyBuilder();
        builder.setSpout("spout", new MySpout());
```

```
        builder.setBolt("add", new MyAddBolt()).shuffleGrouping("spout");
        builder.setBolt("addten",     new     MyAddTenBolt()).shuffleGrouping("add",
"addtenstrim");
        builder.setBolt("out",                                                    new
Print()).shuffleGrouping("addten").shuffleGrouping("add", "printstrim");

        Config conf = new Config();
        conf.setDebug(false);
        LocalCluster cluster = new LocalCluster();
        cluster.submitTopology("WordCount", conf, builder.createTopology());
    }
}
```

Spout 代码定义如下：

```
public class MySpout extends BaseRichSpout {
    private static final long serialVersionUID = 6470207401785986297L;
    private SpoutOutputCollector collector;
    private List<Integer> list=new ArrayList<Integer>();
    private int i=0;

    public MySpout(){
        for(int i=0;i<10;i++){
            list.add(i);
        }
    }
    @SuppressWarnings("rawtypes")
    public void open(Map conf, TopologyContext context,
            SpoutOutputCollector collector) {
        this.collector = collector;
    }

    public void nextTuple() {
        if(i<list.size()){
            int a=list.get(i);
            collector.emit(new Values(a,a),i);
            i++;
        }

    }

    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("a", "b"));
    }
```

```
    public void fail(Object msgId) {
        int in=(Integer) msgId;
        collector.emit(new Values(list.get(in),list.get(in)),in);
    }

}
```

Bolt1 代码定义如下：

```
public class MyAddBolt extends BaseRichBolt{

    private static final long serialVersionUID = 2785733829373403179L;

    private OutputCollector collector;

    @SuppressWarnings("rawtypes")
    public void prepare(Map stormConf, TopologyContext context,
            OutputCollector collector) {
        this.collector=collector;
    }

    public void execute(Tuple input) {
        int a=input.getInteger(0);
        int b=input.getInteger(1);

        int c=a+b;
        if(c>10){
            collector.emit("addtenstrim",input, new Values(c));
        }else{
            collector.emit("printstrim", input, new Values(c));
        }
        collector.ack(input);
    }

    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declareStream("addtenstrim", new Fields("add"));
        declarer.declareStream("printstrim", new Fields("add"));
    }
}
```

Bolt2 代码如下：

```
public class MyAddTenBolt extends BaseRichBolt{

    private static final long serialVersionUID = -917956850826439496L;
```

```
    private OutputCollector collector;

    @SuppressWarnings("rawtypes")
    public void prepare(Map stormConf, TopologyContext context,
            OutputCollector collector) {
        this.collector=collector;
    }

    public void execute(Tuple input) {
        int c=input.getInteger(0);
        c=c+10;
        collector.emit(input,new Values(c));
        collector.ack(input);
    }

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

Bolt3 代码如下：

```
public class Print extends BaseRichBolt{
    private static final long serialVersionUID = -917956850826439496L;

    private OutputCollector collector;

    @SuppressWarnings("rawtypes")
    public void prepare(Map stormConf, TopologyContext context,
            OutputCollector collector) {
        this.collector=collector;
    }

    public void execute(Tuple input) {
        int c=input.getInteger(0);
        System.out.println("最终的输出为："+c);
        if(c==10){
            collector.fail(input);
        }
        collector.ack(input);
    }

    public void declareOutputFields(OutputFieldsDeclarer declarer) {

    }
```

```
}
```

四、安装部署

安装步骤：

- 搭建 Zookeeper 集群；
- 安装 Storm 依赖库，包括 ZeroMQ 2.1.7 、JZMQ 、Java 6 、Python 2.6.6 和 unzip；
- 下载并解压 Storm 发布版本；
- 修改 storm.yaml 配置文件；
- 启动 Storm 各个后台进程；
- 向 storm 集群提交任务；

1、搭建 Zookeeper 集群

Storm 使用 Zookeeper 协调集群，由于 Zookeeper 并不用于消息传递，所以 Storm 给 Zookeeper 带来的压力相当低。大多数情况下，单个节点的 Zookeeper 集群足够胜任，不过为了确保故障恢复或者部署大规模 Storm 集群，可能需要更大规模节点的 Zookeeper 集群（对于 Zookeeper 集群的话，官方推荐的最小节点数为3个）。

2、安装 Storm 依赖库

接下来，需要在 Nimbus 和 Supervisor 机器上安装 Storm 的依赖库，具体如下：

- ZeroMQ 2.1.7
- JZMQ
- Java 6
- Python 2.6.6

以上依赖库的版本是经过Storm测试的，Storm并不能保证在其他版本的Java或Python库下可运行。

安装ZMQ 2.1.7

```
#  wget http://download.zeromq.org/zeromq-2.1.7.tar.gz
#  tar -xzf zeromq-2.1.7.tar.gz
#  cd zeromq-2.1.7
#  ./configure
#  make
#  sudo make install
```

注意事项：

编译可能会出错：configure: error: Unable to find a working C++ compiler，安装一下依赖的rpm包：libstdc++-devel gcc-c++ ，可以上网的情况下：yum install gcc-c++，虚拟机不能上网情况：首先 到 http://mirrors.163.com/centos/6.4/os/x86_64/Packages/ 下载rpm，然后执行如下操作：

```
#  rpm -i libstdc++-devel-4.4.7-3.el6.x86_64.rpm
```

```
#  rpm -i gcc-c++-4.4.7-3.el6.x86_64.rpm

#  rpm -i libuuid-devel-2.17.2-12.9.el6.x86_64.rpm
```

安装 JZMQ

下载后编译安装 JZMQ：

```
#  上传 JZMQ 到服务器

#  cd jzmq

#  ./autogen.sh

#  ./configure

#  make

#  sudo make install
```

在执行./autogen.sh 时，可能会报错:autogen.sh: error: could not find libtool. libtool is required to run autogen.sh. 缺少 libtool 依赖，可以执行 yum install libtool 在线安装，或者手动安装：

```
#  rpm -i autoconf-2.63-5.1.el6.noarch.rpm

#  rpm -i automake-1.11.1-4.el6.noarch.rpm

#  rpm -i libtool-2.2.6-15.5.el6.x86_64.rpm
```

如果运行 ./configure 命令出现问题，参考这里。

安装 Python2.6.6

```
#  tar  - zxvf Python-2.6.6.tgz
```

cd Python-2.6.6

./configure

make

make install

3、下载并解压 Storm 发布版本

下一步，需要在 Nimbus 和 Supervisor 机器上安装 Storm 发行版本。

下载 Storm 发行版本，使用 Storm0.9.3：

上传 storm-0.9.3

unzip storm-0.9.3.zip

4、修改 storm.yaml 配置文件

Storm 发行版本解压目录下有一个 conf/storm.yaml 文件，用于配置 Storm。默认配置在这里可以查看。conf/storm.yaml 中的配置选项将覆盖 defaults.yaml 中的默认配置。以下配置选项是必须在 conf/storm.yaml 中进行配置的：

- ✓ storm.zookeeper.servers: Storm 集群使用的 Zookeeper 集群地址，其格式如下：

  storm.zookeeper.servers:

  - “111.222.333.444″

- “555.666.777.888″

如果 Zookeeper 集群使用的不是默认端口，那么还需要 storm.zookeeper.port 选项。

- ✓ storm.local.dir: Nimbus 和 Supervisor 进程用于存储少量状态，如 jars、confs 等的本地磁盘目录，需要提前创建该目录并给以足够的访问权限。然后在 storm.yaml 中配置该目录，如：storm.local.dir: ″/home/admin/storm/workdir″
- ✓ java.library.path: Storm 使用的本地库（ZMQ 和 JZMQ）加载路径，默认为”/usr/local/lib:/opt/local/lib:/usr/lib”，一般来说 ZMQ 和 JZMQ 默认安装在/usr/local/lib 下，因此不需要配置即可。
- ✓ nimbus.host: Storm 集群 Nimbus 机器地址，各个 Supervisor 工作节点需要知道哪个机器是 Nimbus，以便下载 Topologies 的 jars、confs 等文件，如：nimbus.host: ″111.222.333.444″
- ✓ supervisor.slots.ports: 对于每个 Supervisor 工作节点，需要配置该工作节点可以运行的 worker 数量。每个 worker 占用一个单独的端口用于接收消息，该配置选项即用于定义哪些端口是可被 worker 使用的。默认情况下，每个节点上可运行4个 workers，分别在6700、6701、6702和6703端口，如：

supervisor.slots.ports:

- 6700

- 6701

- 6702

- 6703

5、启动 Storm 各个后台进程

将配置好的 storm，发送给集群中的各个节点。最后一步，启动 Storm 的所有后台进程。和 Zookeeper 一样，Storm 也是快速失败（fail-fast)的系统，这样 Storm 才能在任意时刻被停止，并且当进程重启后被正确地恢复执行。这也是为什么 Storm 不在进程内保存状态的原因,即使 Nimbus 或 Supervisors 被重启,运行中的 Topologies 不会受到影响。

以下是启动 Storm 各个后台进程的方式：

✓ Nimbus: 在 Storm 主控节点上运行”bin/storm nimbus >/dev/null 2>&1 &”启动 Nimbus 后台程序，并放到后台执行；

✓ Supervisor: 在 Storm 各个工作节点上运行”bin/storm supervisor >/dev/null 2>&1 &”启动 Supervisor 后台程序，并放到后台执行；

✓ UI: 在 Storm 主控节点上运行”bin/storm ui >/dev/null 2>&1 &" 启动 UI 后台程序,并放到后台执行,启动后可以通过 http://{nimbus

host}:8080观察集群的worker资源使用情况、Topologies的运行状态等信息。

注意事项：

- ✓ 启动Storm后台进程时，需要对conf/storm.yaml配置文件中设置的storm.local.dir目录具有写权限。
- ✓ Storm后台进程被启动后，将在Storm安装部署目录下的logs/子目录下生成各个进程的日志文件。
- ✓ 经测试，Storm UI必须和Storm Nimbus部署在同一台机器上，否则UI无法正常工作，因为UI进程会检查本机是否存在Nimbus链接。

至此，Storm集群已经部署、配置完毕，可以向集群提交拓扑运行了。

6、向集群提交任务

- ✓ 启动Storm Topology：

```
storm jar allmycode.jar org.me.MyTopology arg1 arg2 arg3
```

其中，allmycode.jar是包含Topology实现代码的jar包，org.me.MyTopology的main方法是Topology的入口，arg1、arg2和arg3为org.me.MyTopology执行时需要传入的参数。

- ✓ 停止Storm Topology：

storm kill {toponame}

其中，{toponame}为 Topology 提交到 Storm 集群时指定的 Topology 任务名称。

五、消息的可靠处理

1、简介

storm 可以确保 spout 发送出来的每个消息都会被完整的处理。本章将会描述 storm 体系是如何达到这个目标的，并将会详述开发者应该如何使用 storm 的这些机制来实现数据的可靠处理。

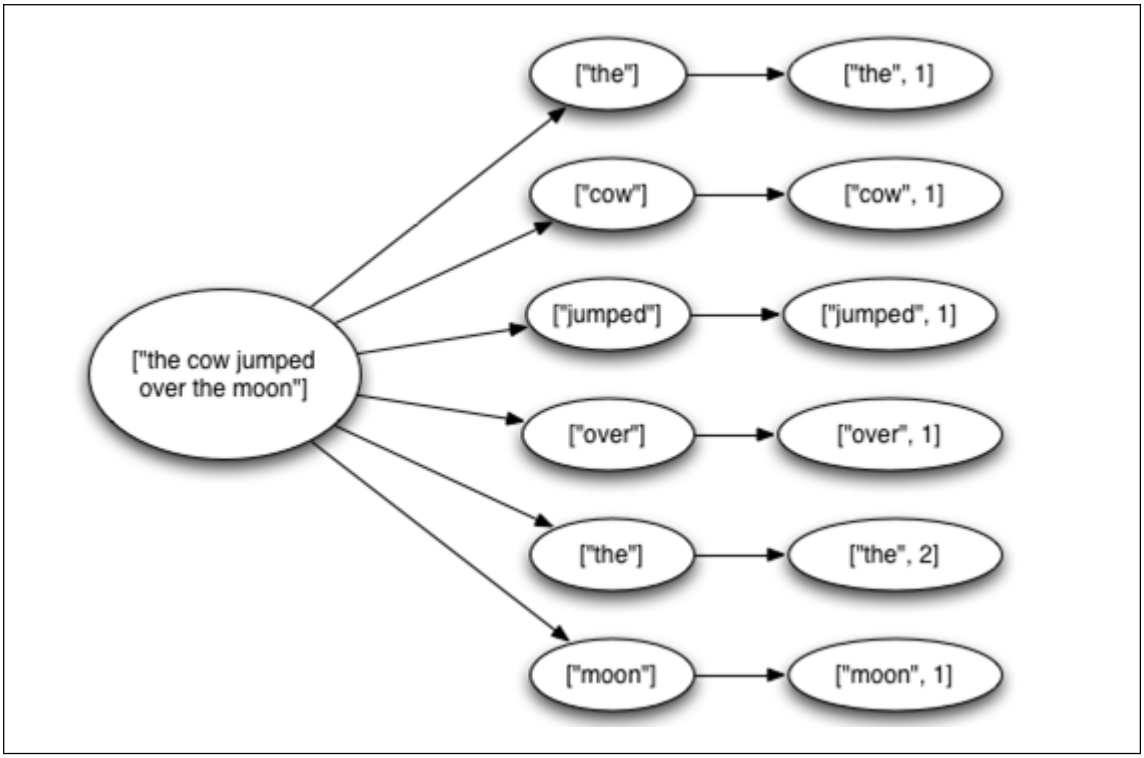
2、理解消息被完整处理

一个消息(tuple)从 spout 发送出来，可能会导致成百上千的消息基于此消息被创建。

我们来思考一下流式的“单词统计”的例子：storm 任务从数据源每次读取一个完整的英文句子；将这个句子分解为独立的单词，最后，实时的输出每个单词以及它出现过的次数。

本例中，每个从 spout 发送出来的消息（每个英文句子）都会触发很多的消息被创建，那些从句子中分隔出来的单词就是被创建出来的新消息。

这些消息构成一个树状结构，我们称之为“tuple tree”，看起来如图所示：



在什么条件下，Storm 才会认为一个从 spout 发送出来的消息被完整处理呢？答案就是下面的条件同时被满足：

- ✓ tuple tree 不再生长
- ✓ 树中的任何消息被标识为“已处理”

如果在指定的时间内，一个消息衍生出来的 tuple tree 未被完全处理成功，则认为此消息未被完整处理。这个超时值可以通过任务级参数 Config.TOPOLOGY_MESSAGE_TIMEOUT_SECS 进行配置，默认超时值为30秒。

3、消息的生命周期

如果消息被完整处理或者未被完整处理，Storm 会如何进行接下来的操作呢？为了弄清这个问题，我们来研究一下从 spout 发出来的消息的生命周期。这里列出了 spout 应该实现的接口：

```
public interface ISpout extends Serializable {

    void open(Map conf, TopologyContext context, SpoutOutputCollector collector);

    void close();

    void activate();

    void deactivate();

    void nextTuple();

    void ack(Object msgId);

    void fail(Object msgId);
}
```

Spout 方法的调用顺序为：

- ✓ declareOutputFields()
- ✓ open()
- ✓ activate()
- ✓ nextTuple()　　（循环调用 ）
- ✓ deactivate()

首先， Storm 使用 spout 实例的 nextTuple()方法从 spout 请求一个消息（tuple）。 收到请求以后，spout 使用 open 方法中提供的 SpoutOutputCollector 向它的输出流发送一个或多个消息。每发送一个消息，Spout 会给这个消息提供一个 message ID，它将会被用来标识这个消息。SpoutOutputCollector 中发送消息格式如下：collector.emit(new Values(sentence), "messageId")。

接来下，这些消息会被发送到后续业务处理的 bolts，并且 Storm 会跟踪由此消息产生出来的新消息。当检测到一个消息衍生出来的

tuple tree 被完整处理后，Storm 会调用 Spout 中的 ack 方法，并将此消息的 messageID 作为参数传入。同理，如果某消息处理超时，则此消息对应的 Spout 的 fail 方法会被调用，调用时此消息的 messageID 会被作为参数传入。

注意：一个消息只会由发送它的那个 spout 任务来调用 ack 或 fail。如果系统中某个 spout 由多个任务运行，消息也只会由创建它的 spout 任务来应答（ack 或 fail），决不会由其他的 spout 任务来应答。

当 KestrelSpout 从 kestrel 队列中读取一个消息，表示它“打开”了队列中某个消息。这意味着，此消息并未从队列中真正的删除，而是将此消息设置为“pending”状态，它等待来自客户端的应答，被应答以后，此消息才会被真正的从队列中删除。处于“pending”状态的消息不会被其他的客户端看到。另外，如果一个客户端意外的断开连接，则由此客户端“打开”的所有消息都会被重新加入到队列中。当消息被“打开”的时候，kestrel 队列同时会为这个消息提供一个唯一的标识。

KestrelSpout 就是使用这个唯一的标识作为这个 tuple 的 messageID 的。稍后当 ack 或 fail 被调用的时候，KestrelSpout 会把 ack 或者 fail 连同 messageID 一起发送给 kestrel 队列，kestrel 会将消息从队列中真正删除或者将它重新放回队列中。

4、可靠相关的 API

为了使用 Storm 提供的可靠处理特性，我们需要做两件事情：

- ✓ 无论何时在 tuple tree 中创建了一个新的节点，我们需要明确的通知 Storm：collector.emit(tuple,new Values(word))；
- ✓ 当处理完一个单独的消息时，我们需要告诉 Storm 这棵 tuple tree 的变化状态：collector.ack(tuple)；

上面的这两步一般都是在 bolt 中调用。通过上面的两步，storm 就可以检测到一个 tuple tree 何时被完全处理了，并且会调用相关的 ack 或 fail 方法。Storm 提供了简单明了的方法来完成上述两步。

为 tuple tree 中指定的节点增加一个新的节点，我们称之为锚定（anchoring）。锚定是在我们发送消息的同时进行的。为了更容易说明问题，我们使用下面代码作为例子。本示例的 bolt 将包含整句话的消息分解为一系列的子消息，每个子消息包含一个单词。

```
public class SplitSentence extends BaseRichBolt {
        OutputCollector _collector;

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

        public void execute(Tuple tuple) {
            String sentence = tuple.getString(0);
            for(String word: sentence.split(" ")) {
                _collector.emit(tuple, new Values(word));
            }
            _collector.ack(tuple);
        }

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

每个消息都通过这种方式被锚定：把输入消息作为 emit 方法的第一个参数。因为 word 消息被锚定在了输入消息上，这个输入消息是 spout 发送过来的 tuple tree 的根节点，如果任意一个 word 消息处理失败，派生这个 tuple tree 那个 spout 消息将会被重新发送。

与此相反，我们来看看使用下面的方式 emit 消息时，Storm 会如何处理：collector.emit(new Values(word));

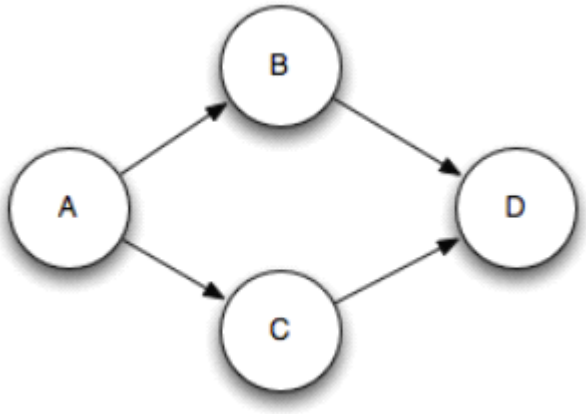
如果以这种方式发送消息，将会导致这个消息不会被锚定。如果此 tuple tree 中的消息处理失败，派生此 tuple tree 的根消息不会被重新发送。根据任务的容错级别，有时候很适合发送一个非锚定的消息。

一个输出消息可以被锚定在一个或者多个输入消息上，这在做 join 或聚合的时候是很有用的。一个被多重锚定的消息处理失败，会导致与之关联的多个 spout 消息被重新发送。多重锚定通过在 emit 方法中指定多个输入消息来实现：

```
List<Tuple> anchors = new ArrayList<Tuple>();
anchors.add(tuple1);
anchors.add(tuple2);
_collector.emit(anchors, new Values(1, 2, 3));
```

多重锚定会将被锚定的消息加到多棵 tuple tree 上。

注意：多重绑定可能会破坏传统的树形结构，从而构成一个 DAGs（有向无环图），如图所示：



很多 bolt 遵循特定的处理流程： 读取一个消息、发送它派生出来的子消息、在 execute 结尾处应答此消息。一般的过滤器（filter）或者是简单的处理功能都是这类的应用。Storm 有一个 IBasicBolt 接口封装了上述的流程。示例 SplitSentence 可以使用 IBasicBolt 来重写：

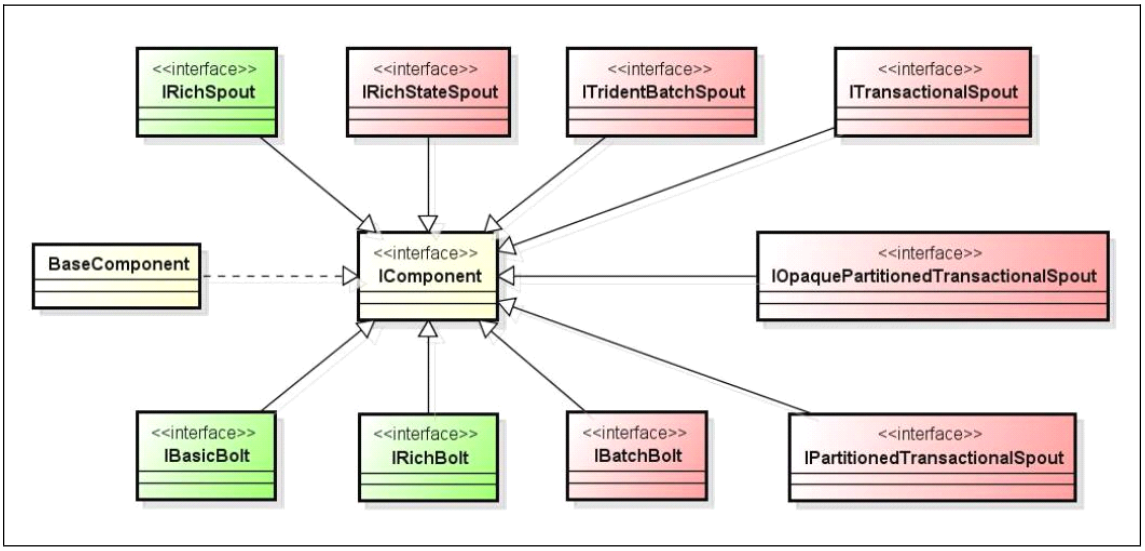
```
public class SplitSentence extends BaseBasicBolt {
        public void execute(Tuple tuple, BasicOutputCollector collector) {
            String sentence = tuple.getString(0);
            for(String word: sentence.split(" ")) {
                collector.emit(new Values(word));
            }
        }

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

使用这种方式，代码比之前稍微简单了一些，但是实现的功能是一样的。发送到 BasicOutputCollector 的消息会被自动的锚定到输入消息，并且当 execute 执行完毕的时候，会自动的调用 spout 的 ack 方法。但执行某个操作失败的时候，需要抛出 FailException 异常才会自动调用 spout 的 fail 方法。

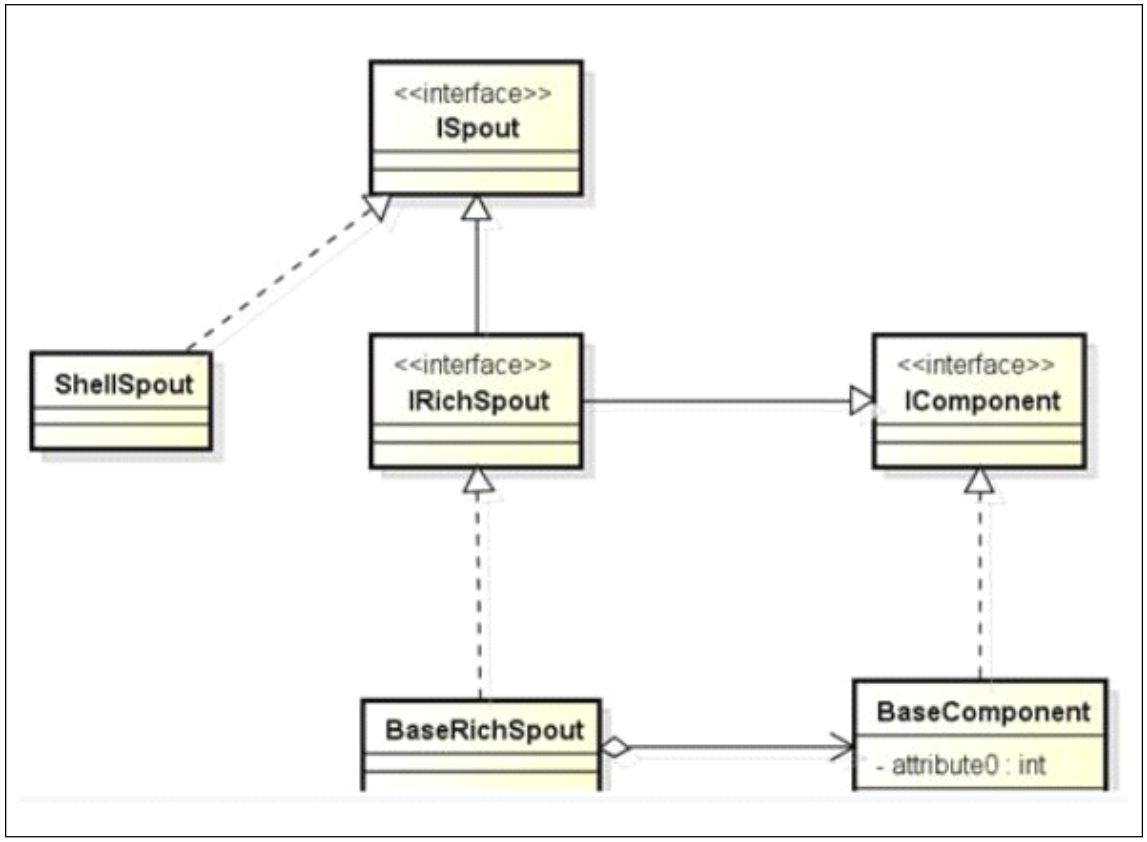
很多情况下，一个消息需要延迟应答，例如聚合或者是 join。只有根据一组输入消息得到一个结果之后，才会应答之前所有的输入消息。并且聚合和 join 大部分时候对输出消息都是多重锚定。然而，这些特性不是 IBasicBolt 所能处理的，必须使用 IRichBolt 自己手动控制。

Storm 提供的 spout 和 bolt 类图关系如下：

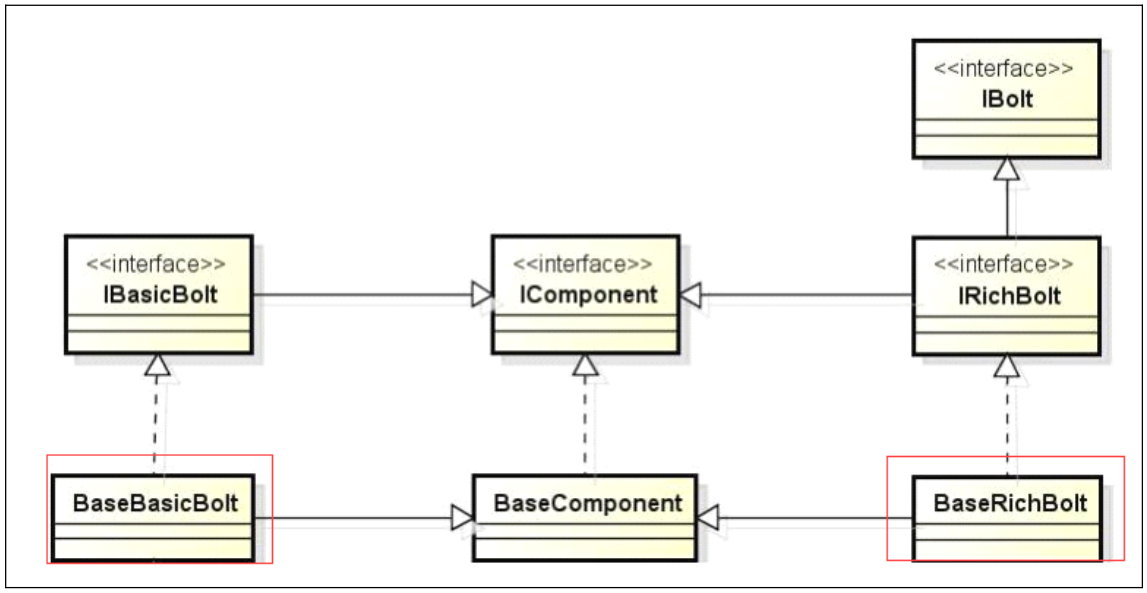


绿色是我们最常用的，红色是与事务相关的。

其中，常用的 spout（IRichSpout）类结构图如下：



常用的 bolt 类图如下所示：



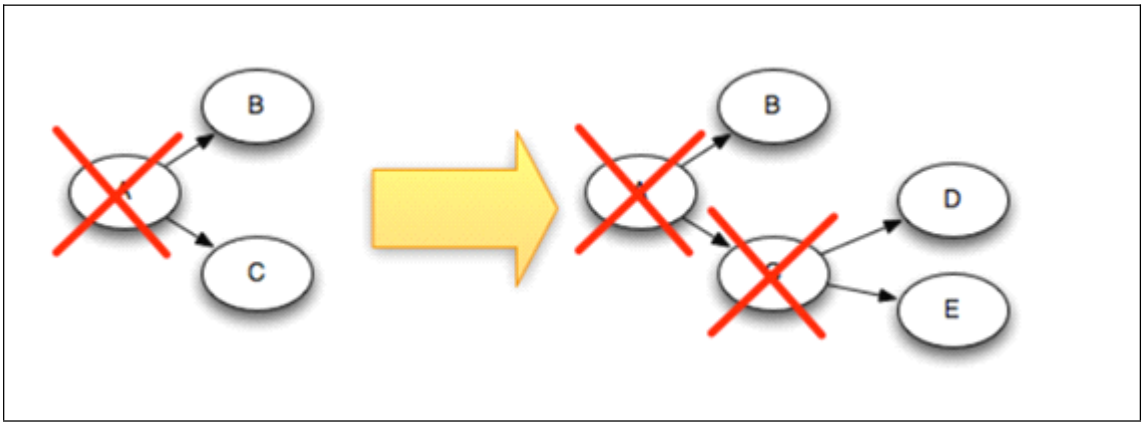
5、storm 可靠性的背后

Storm 系统中有一组叫做“acker”的特殊的任务，它们负责跟踪 DAG（有向无环图）中的每个消息。每当发现一个 DAG 被完全处理，

它就向创建这个根消息的 spout 任务发送一个信号。拓扑中 acker 任务的并行度可以通过配置参数 Config.TOPOLOGY_ACKERS 来设置。默认的 acker 任务并行度为1，当系统中有大量的消息时，应该适当提高 acker 任务的并发度。

每当 bolt 新生成一个消息，对应 tuple tree 中的根消息的 messageId 就拷贝到这个消息中。当这个消息被应答的时候，它就把关于 tuple tree 变化的信息发送给跟踪这棵树的 acker。例如，他会告诉 acker：本消息已经处理完毕，但是我派生出了一些新的消息，帮忙跟踪一下吧。

举个例子，假设消息 D 和 E 是由消息 C 派生出来的，这里演示了消息 C 被应答时，tuple tree 是如何变化的。



因为在 C 被从树中移除的同时 D 和 E 会被加入到 tuple tree 中，因此 tuple tree 不会被过早的认为已完全处理。

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。

6、选择合适的可靠性级别

Acker 任务是轻量级的，所以在拓扑中并不需要太多的 acker 存在。可以通过 Storm UI 来观察 acker 任务的吞吐量，如果看上去吞吐量不够的话，说明需要添加额外的 acker。

有三种方法可以控制消息的可靠处理机制：

- ✓ 将参数 Config.TOPOLOGY_ACKERS 设置为 0(`config.setNumAckers(0)`)，通过此方法，当 Spout 发送一个消息的时候，它的 ack 方法将立刻被调用，这样就不能保证消息的可靠性；
- ✓ Spout 发送一个消息时，不指定此消息的 messageID。这样 spout 中的 ack 和 fail 方法将不会被调用，即使 acker 参数不为0，但这样就不能控制失败后是否重发消息。即在 spout 的 nextTuple 方法中这样发射数据：collector.emit(new Values(1));
- ✓ 如果你不在意某个消息派生出来的子孙消息的可靠性，则此消息派生出来的子消息在发送时不要做锚定，即在 emit 方法中不指定输入消息。因为这些子孙消息没有被锚定在任何 tuple tree 中，因此他们的失败不会引起任何 spout 重新发送消息。

如果需要消息的可靠处理，需要如下条件同时满足：

- ✓ 设置 Ackers 的数量为非0；conf.setNumAckers(1);
- ✓ 在 spout 发射消息时，绑定 messageId；collector.emit(new Values(1), messageId);

- ✓ 在 bolt 中发射消息时，需要锚定到上一个消息上；
  collector.emit(input, new Values(word));

7、集群的各级容错

到现在为止，大家已经理解了 Storm 的可靠性机制，并且知道了如何选择不同的可靠性级别来满足需求。接下来我们研究一下 Storm 如何保证在各种情况下确保数据不丢失。

1、任务级失败

- ✓ bolt 任务失败。此时，acker 中所有与此 bolt 任务关联的消息都会因为超时而失败，对应 spout 的 fail 方法将被调用。
- ✓ acker 任务失败。如果 acker 任务本身失败了，它在失败之前持有的所有消息都将会因为超时而失败。Spout 的 fail 方法将被调用。
- ✓ Spout 任务失败。这种情况下，Spout 任务对接的外部设备（如 MQ）负责消息的完整性。例如当客户端异常的情况下，kestrel 队列会将处于 pending 状态的所有的消息重新放回到队列中。其他的 spout 数据源，可能需要我们自行维护这个消息的完整性。

2、任务槽(slot) 故障

- ✓ worker 失败。每个 worker 中包含数个 bolt（或 spout）任务。supervisor 负责监控这些任务，当 worker 失败后，supervisor 会尝试在本机重启它。
- ✓ supervisor 失败。supervisor 是无状态的，因此 supervisor 的失败不会影响当前正在运行的任务，只要及时的将它重新启动即可。**supervisor 不是自举的，需要外部监控来及时重启。**
- ✓ nimbus 失败。nimbus 是无状态的，因此 nimbus 的失败不会影响当前正在运行的任务（nimbus 失败时，无法提交新的任务），只要及时的将它重新启动即可。**nimbus 不是自举的，需要外部监控来及时重启。**

3、集群节点（机器）故障

- ✓ storm 集群中的节点故障。此时 nimbus 会将此机器上所有正在运行的任务转移到其他可用的机器上运行。
- ✓ zookeeper 集群中的节点故障。zookeeper 保证少于半数的机器宕机仍可正常运行，及时修复故障机器即可。