一. Storm简介

Storm是一个分布式的、高容错的实时计算系统。

Storm对于实时计算的的意义相当于Hadoop对于批处理的意义。Hadoop为我们提供了Map和Reduce原语，使我们对数据进行批处理变的非常的简单和优美。同样，Storm也对数据的实时计算提供了简单Spout和Bolt原语。

Storm适用的场景
1. 流数据处理：Storm可以用来用来处理源源不断的消息，并将处理之后的结果保存到持久化介质中。
2. 分布式RPC：由于Storm的处理组件都是分布式的，而且处理延迟都极低，所以可以Storm可以做为一个通用的分布式RPC框架来使用。

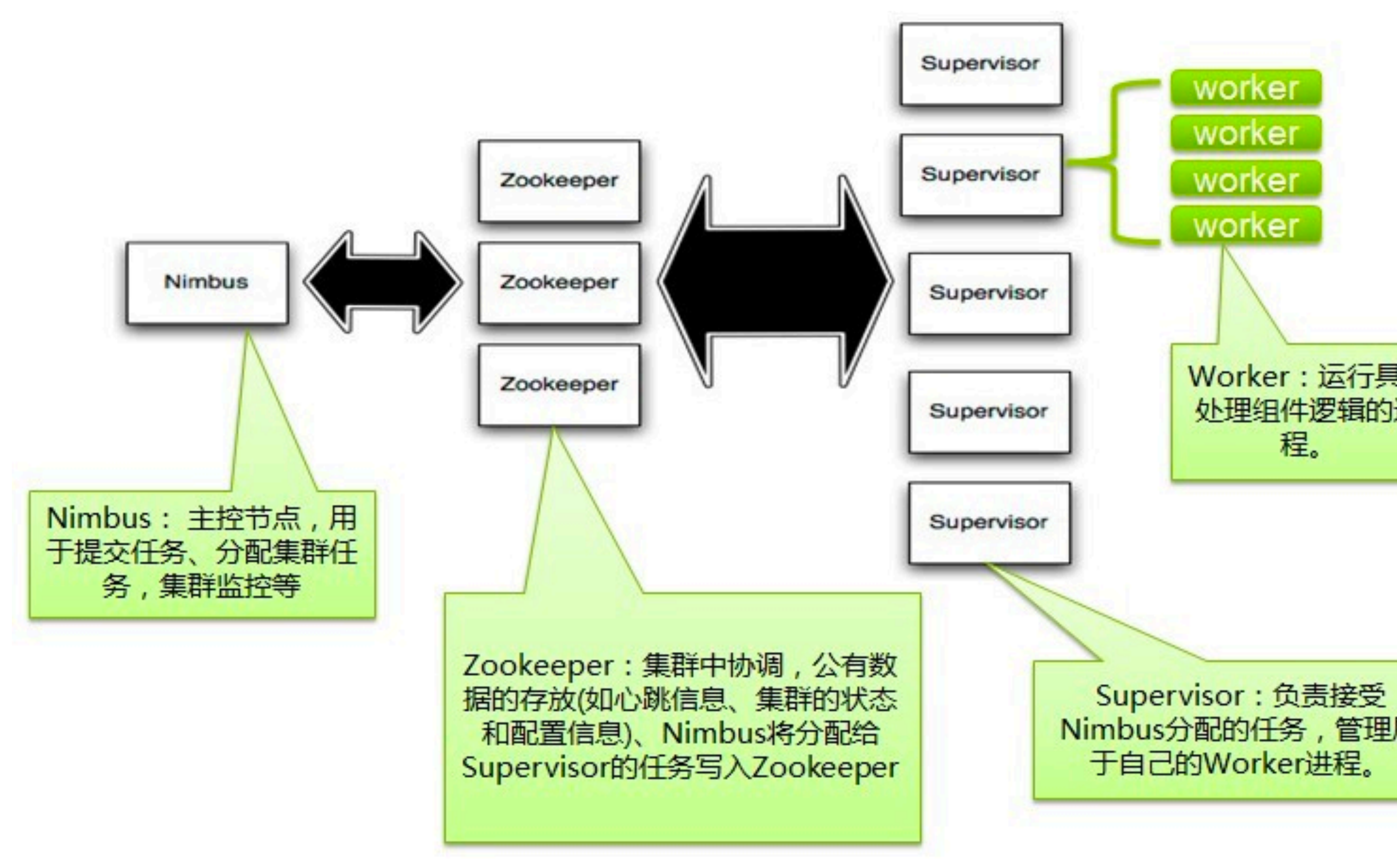
二. Storm集群基本组件

Storm的集群表面上看和hadoop的集群非常像。但是在Hadoop上面你运行的是MapReduce的Job, 而在Storm上面你运行的是Topology。它们是非常不一样的，一个关键的区别是： 一个MapReduce Job最终会结束， 而一个Topology永远运行（除非你显式的杀掉他）。

在Storm的集群里面有两种节点：

1. 控制节点(master node)：控制节点上面运行一个后台程序Nimbus， 它的作用类似Hadoop里面的JobTracker。Nimbus负责在集群里面分发代码，分配工作给机器， 并且监控状态。

2. 工作节点(worker node)：每一个工作节点上面运行一个叫做Supervisor的进程（类似 TaskTracker）。Supervisor会监听分配给它那台机器的工作，根据需要 启动/关闭工作进程。每一个工作进程执行一个Topology（类似 Job）的一个子集；一个运行的Topology由运行在很多机器上的很多工作进程 Worker（类似 Child）组成。



Nimbus和Supervisor之间的所有协调工作都是通过一个Zookeeper集群来完成。并且，nimbus进程和supervisor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，它们可以继续工作， 就好像什么都没有发生过似的。这个设计使得storm不可思议的稳定。

1. Topology

为了在storm上面做实时计算， 你要去建立一些topology。一个topology就是一个计算节点所组成的图，Topology里面的每个处理节点都包含处理逻辑， 而节点之间的连接则表示数据流动的方向。

运行一个Topology是很简单的。

- 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：strom jar all-your-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文件。

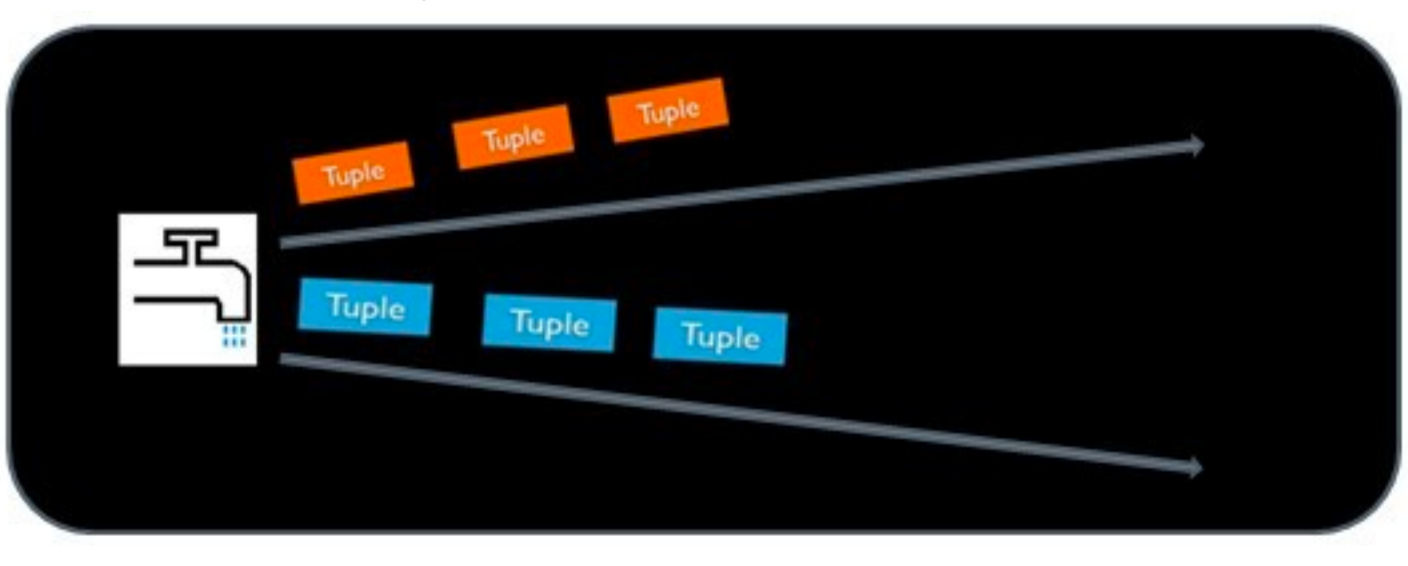
因为topology的定义其实就是一个Thrift结构并且nimbus就是一个Thrift服务， 可以用任何语言创建并且提交topology。

2. Stream

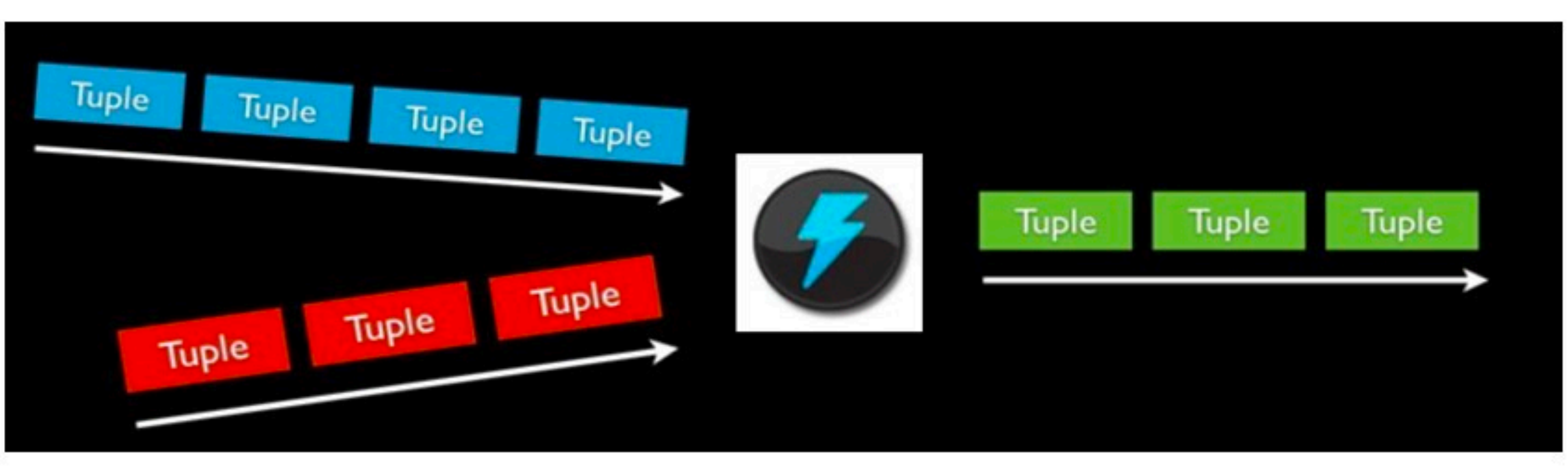
Stream是storm里面的关键抽象。

一个stream是一个没有边界的tuple序列。storm提供一些原语来分布式地、可靠地把一个stream传输进一个新的stream。比如： 你可以把一个tweets流传输到热门话题的流。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对应的接口以处理你的应用的逻辑。

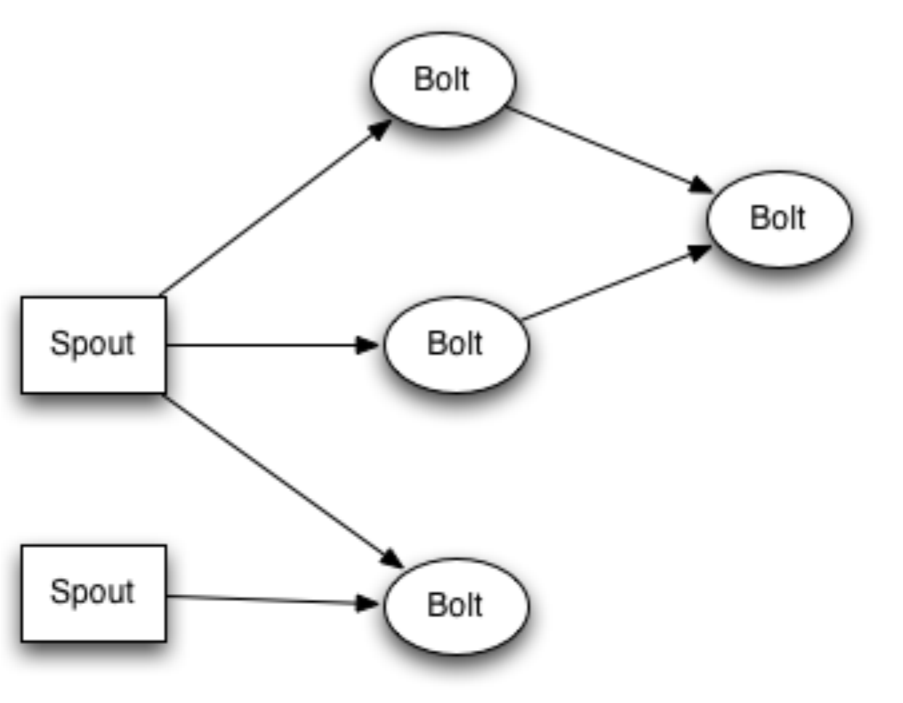
spout是流的源头。比如一个spout可能从Kafka队列里面读取消息并且把这些消息发射成一个流。又比如一个spout可以调用twitter的一个api并且把返回的tweets发射成一个流。通常Spout会从外部数据源（队列、数据库等）读取数据，然后封装成Tuple形式，之后发送到Stream中。Spout是一个主动的角色，在接口内部有个nextTuple函数，Storm框架会不停的调用该函数。



bolt可以接收任意多个输入stream， 作一些处理， 有些bolt可能还会发射一些新的stream。一些复杂的流转换， 比如从一些tweet里面计算出热门话题，需要多个步骤，从而也就需要多个bolt。 Bolt可以做任何事情: 运行函数，过滤tuple，做一些聚合， 做一些合并以及访问数据库等等。Bolt处理输入的Stream，并产生新的输出Stream。Bolt是一个被动的 角色，其接口中有一个execute(Tuple input)方法，在接收到消息之后会调用此函数，用户可以在此方法中执行自己的处理逻辑。



spout和bolt所组成一个网络会被打包成topology， topology是storm里面最高一级的抽象（类似 Job），可以把topology提交给storm的集群来运行。



3. 数据模型

storm使用tuple来作为它的数据模型。每个tuple是一堆值，每个值有一个名字，并且每个值可以是任何类型， 在我的理解里面一个tuple可以看作一个没有方法的java对象。总体来看，storm支持所有的基本类型、字符串以及字节数组作为tuple的值类型。你也可以使用你自己定义的类型来作为值类型， 只要你实现对应的序列化器(serializer)。

一个Tuple代表数据流中的一个基本的处理单元，例如一条cookie日志，它可以包含多个Field，每个Field表示一个属性。

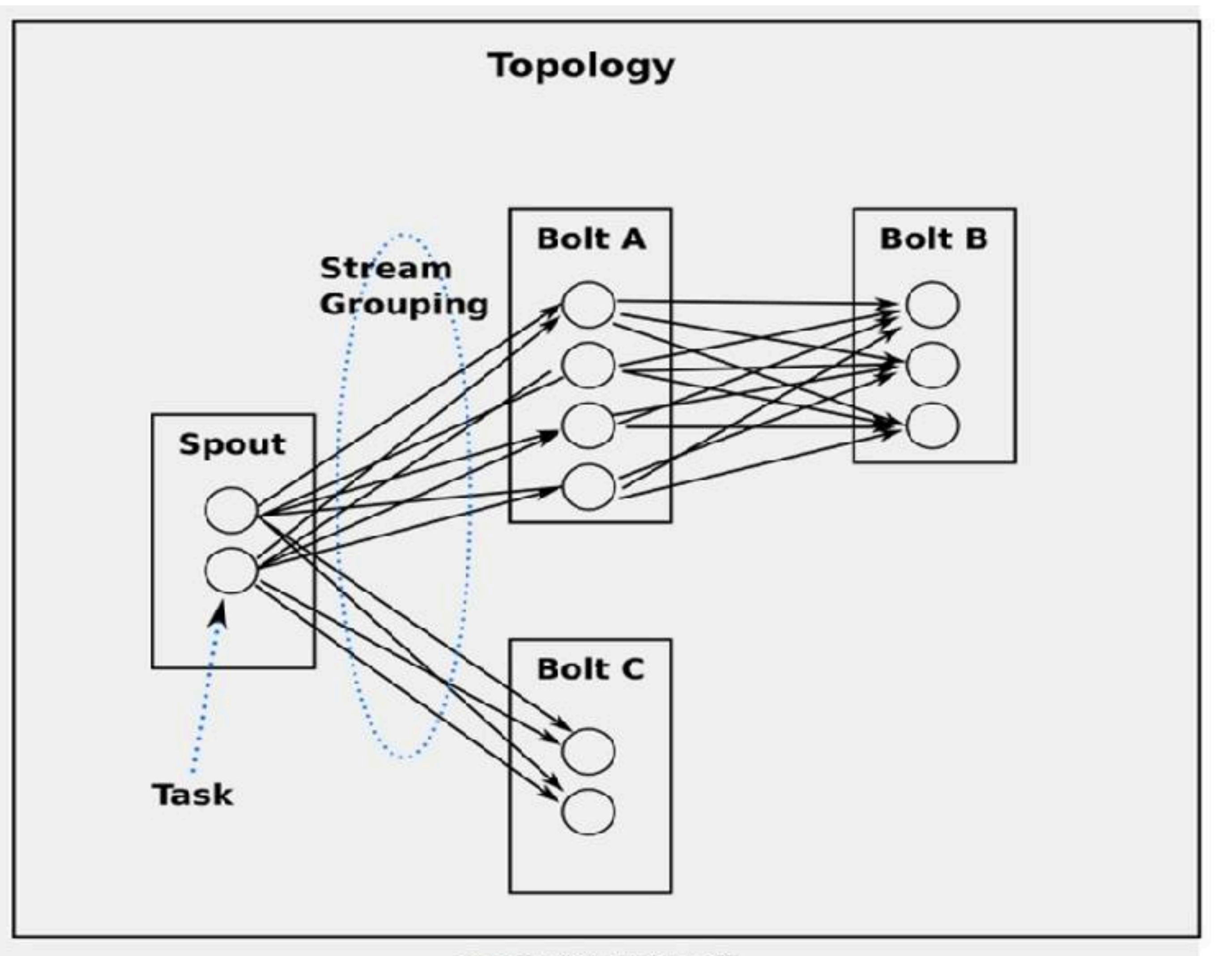
Tuple本来应该是一个Key-Value的Map，由于各个组件间传递的tuple的字段名称已经事先定义好了，所以Tuple只需要按序填入各个Value，所以就是一个Value List。一个没有边界的、源源不断的、连续的Tuple序列就组成了Stream。

topology里面的每个节点必须定义它要发射的tuple的每个字段。 比如下面这个bolt定义它所发射的tuple包含两个字段，类型分别是: double和triple。

```
public class DoubleAndTripleBolt implements IRichBolt {
    privateOutputCollectorBase _collector;

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

    @Override
    public void execute(Tuple input) {
        intval = input.getInteger(0);
        _collector.emit(input,newValues(val*2, val*3));
        _collector.ack(input);
    }

    @Override
    public void cleanup() {
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(newFields("double","triple"));
    }
}
```

4. 流分组策略

流分组策略告诉topology如何在两个组件之间发送tuple。 要记住， spouts和bolts以很多task的形式在topology里面同步执行。

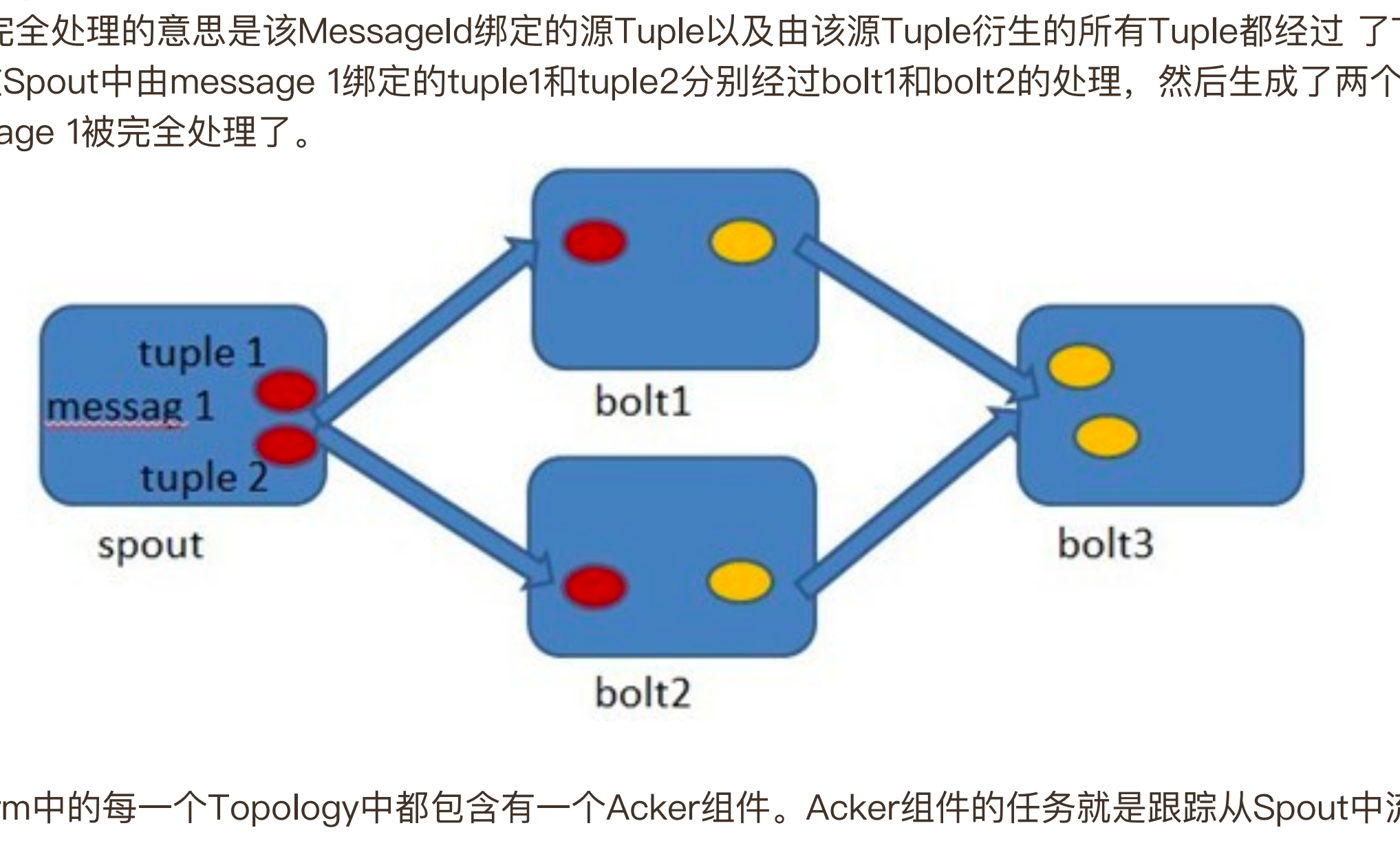


storm里面各个对象的示意图

5. 可靠的消息处理

Storm允许用户在Spout中发射一个新的源Tuple时为其指定一个MessageId，这个MessageId可以是任意的Object对象。多个源Tuple可以共用同一个MessageId，表示这多个源Tuple对用户来说是同一个消息单元。Storm的可靠性是指Storm会告知用户每一个消息单元是否在一个指定的时间内被完全处理。完全处理的意思是该MessageId绑定的源Tuple以及由该源Tuple衍生的所有Tuple都经过 了Topology中每一个应该到达的Bolt的处理。

在Spout中由message 1绑定的tuple1和tuple2分别经过bolt1和bolt2的处理，然后生成了两个新的Tuple，并最终流向了bolt3。当bolt3处理完之后，称message 1被完全处理了。



Storm中的每一个Topology中都包含有一个Acker组件。Acker组件的任务就是跟踪从Spout中流出的每一个messageId所绑定的Tuple树中的所有Tuple的处理情况。如果在用户设置的最大超时时间内这些Tuple没有被完全处理，那么Acker会告诉Spout该消息处理失败，相反则会告知Spout该消息处理成功。

Acker是如何记录Tuple的处理结果呢?

- A xor A = 0.
- A xor B…xor B xor A = 0，其中每一个操作数出现且仅出现两次。

在Spout中，Storm系统会为用户指定的MessageId生成一个对应的64位的整数，作为整个Tuple Tree的RootId。RootId会被传递给Acker以及后续的Bolt来作为该消息单元的唯一标识。同时，无论Spout还是Bolt每次新生成 一个Tuple时，都会赋予该Tuple一个唯一的64位整数的Id。

当Spout发射完某个MessageId对应的源Tuple之后，它会告诉Acker自己发射的RootId以及生成的那些源Tuple的Id。而当 Bolt处理完一个输入Tuple并产生出新的Tuple时，也会告知Acker自己处理的输入Tuple的Id以及新生成的那些Tuple的Id。 Acker只需要对这些Id进行异或运算，就能判断出该RootId对应的消息单元是否成功处理完成了。

三. 简单的Topology

让我们来看一个简单的topology的例子， 我们看一下storm-starter里面的ExclamationTopology

```
TopologyBuilder builder = newTopologyBuilder();
builder.setSpout(1,newTestWordSpout(),10);
builder.setBolt(2,newExclamationBolt(),3).shuffleGrouping(1);
builder.setBolt(3,newExclamationBolt(),2).shuffleGrouping(2);
```

setBolt方法返回一个InputDeclarer对象， 这个对象是用来定义Bolt的输入。 这里第一个Bolt声明它要读取spout所发射的所有的tuple使用shuffle grouping。而第二个bolt声明它读取第一个bolt所发射的tuple。shuffle grouping表示所有的tuple会被随机的分发给bolt的所有task。

如果你想第二个bolt读取spout和第一个bolt所发射的所有的tuple， 那么你应该这样定义第二个bolt:

```
builder.setBolt(3,newExclamationBolt(),5)
              .shuffleGrouping(1)
              .shuffleGrouping(2);
```

storm的运行有两种模式: 本地模式和分布式模式

1. 本地模式中， 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。

2. 分布式模式下， storm由一堆机器组成。当你提交topology给master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。

下面是本地模式运行Strom程序

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

LocalCluster cluster = newLocalCluster();
cluster.submitTopology("test", conf, builder.createTopology());
Utils.sleep(10000);
cluster.killTopology("test");
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。

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. topology里面的每个组件会被需要线程来执行。每个组件到底用多少个线程是通过setBolt和setSpout来指定的。这些线程都运行在工作进程里面. 每一个工作进程包含一些节点的一些工作线程。
- 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。