Storm 简介

- 场景

伴随着信息科技日新月异的发展，信息呈现出爆发式的膨胀，人们获取信息的途径也更加多样、更加便捷，同时对于信息的时效性要求也越来越高。举个搜索场景中的例子，当一个卖家发布了一条宝贝信息时，他希望的当然是这个宝贝马上就可以被卖家搜索出来、点击、购买啦，相反，如果这个宝贝要等到第二天或者更久才可以被搜出来，估计这个大哥就要骂娘了。再举一个推荐的例子，如果用户昨天在淘宝上买了一双袜子，今天想买一副泳镜去游泳，但是却发现系统在不遗余力地给他推荐袜子、鞋子，根本对他今天寻找泳镜的行为视而不见，估计这哥们心里就会想推荐你妹呀。再比如交通路况的实时播报系统，如果一个地方发生拥堵，信息传送过去是 30 秒钟，但是计算机却花了 10 分钟才把计算结果反馈回来，而这时却发现这个地方却早已不赌塞了，这样的情况其实是我们无法容忍的。其实稍微了解点背景知识的码农们都知道，这是因为后台系统做的是每天一次的全量处理，而且大多是在夜深人静之时做的，那么你今天白天做的事情当然要明天才能反映出来啦。

- 实现一个实时计算系统

全量数据处理使用的大多是鼎鼎大名的 hadoop+hive，作为一个批处理系统，hadoop 以其吞吐量大、自动容错等优点，在海量数据处理上得到了 广泛的使用。但是，hadoop 不擅长实时计算，因为它天然就是为批处理而生的，这也是大数据行业的一个共识。否则最近这两年也不会有 s4,storm,puma 这些实时计算系统如雨后春笋般冒出来啦。先抛开 s4,storm,puma 这些系统不谈，我们首先来看一下，如果让我们自己设计一个实时计算系统，我们要解决哪些问题。

1. 低延迟。都说了是实时计算系统了，延迟是一定要低的。
2. 高性能。性能不高就是浪费机器，浪费机器是要受批评的哦。
3. 分布式。系统都是为应用场景而生的，如果你的应用场景、你的数据和计算单机就能搞定，那么不用考虑这些复杂的问题了。我们所说的是单机搞不定的情况。
4. 可扩展。伴随着业务的发展，我们的数据量、计算量可能会越来越大，所以希望这个系统是可扩展的。
5. 容错。这是分布式系统中通用问题。一个节点挂了不能影响我的应用。

如果仅仅需要解决这 5 个问题，可能会有无数种方案，而且各有千秋，随便举一种方案，使用消息队列+分布在各个机器上的工作进程就可以啦。但是如果我们同时还需要保证下面的这几个问题呢？

1. 容易在上面开发应用程序。你设计的系统需要应用程序开发人员考虑各个处理组件的分布、消息的传递吗？如果是，那有点麻烦啊，开发人员可能会用不好，也不会想去用。
2. 消息不丢失。用户发布的一个宝贝消息不能在实时处理的时候给丢了，对吧？更严格一点，如果是一个精确数据统计的应用，那么它处理的消息要不多不少才行。
3. 消息严格有序。有些消息之间是有强相关性的，比如同一个宝贝的更新和删除操作消息，如果处理时搞乱顺序完全是不一样的效果了。

不知道大家对这些问题是否都有了自己的答案，如果还没有，下面让我们带着这些问题，一起来看一看 storm 的解决方案吧。

- Storm 是什么

如果只用一句话来描述 storm 的话，可能会是这样：分布式实时计算系统。按照 storm 作者的说法，storm 对于实时计算的意义类似于 hadoop 对于批处理的意义。我们都知道，根据 google mapreduce 来实现的 hadoop 为我们提供了 map, reduce 原语，使我们的批处理程序变得非常地简单和优美。同样，storm 也为实时计算提供了一些简单优美的原语。后面再详细介绍。

现在，我们来看一下 storm 的适用场景。

1. 流数据处理。Storm 可以用来处理源源不断流进来的消息，处理之后将结果写入到某个存储中去。
2. 分布式 rpc。由于 storm 的处理组件是分布式的，而且处理延迟极低，所以可以作为一个通用的分布式 rpc 框架来使用。当然，其实我们的搜索引擎本身也是一个分布式 rpc 系统。

说了半天，好像都是很玄乎的东西，下面我们开始具体讲解 storm 的基本概念和它内部的一些实现原理吧。

- Storm 的基本概念

首先我们通过一个 storm 和 hadoop 的对比来了解 storm 中的基本概念。

	Hadoop	Storm
系统角色	JobTracker	Nimbus
	TaskTracker	Supervisor
	Child	Worker
应用名称	Job	Topology
组件接口	Mapper/Reducer	Spout/Bolt

表 3-1

接下来我们再来具体看一下这些概念。

1. Nimbus：负责资源分配和任务调度。
2. Supervisor：负责接受 nimbus 分配的任务，启动和停止属于自己管理的 worker 进程。
3. Worker：运行具体处理组件逻辑的进程。
4. Task：worker 中每一个 spout/bolt 的线程称为一个 task. 在 storm0.8 之后，task 不再与物理线程对应，同一个 spout/bolt 的 task 可能会共享一个物理线程，该线程称为 executor。

下面这个图描述了以上几个角色之间的关系。

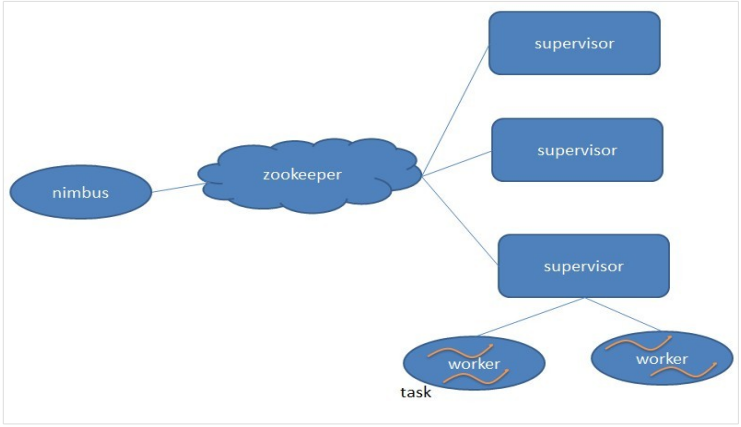

图 3-1

1. Topology：storm 中运行的一个实时应用程序，因为各个组件间的消息流动形成逻辑上的一个拓扑结构。
2. Spout：在一个 topology 中产生源数据流的组件。通常情况下 spout 会从外部数据源中读取数据，然后转换为 topology 内部的源数据。 Spout 是一个主动的角色，其接口中有个 nextTuple()函数，storm 框架会不停地调用此函数，用户只要在其中生成源数据即可。

3. Bolt：在一个 topology 中接受数据然后执行处理的组件。Bolt 可以执行过滤、函数操作、合并、写数据库等任何操作。Bolt 是一个被动的角 色，其接口中有个 execute(Tuple input)函数,在接受到消息后会调用此函数，用户可以在其中执行自己想要的操作。
4. Tuple：一次消息传递的基本单元。本来应该是一个 key-value 的 map，但是由于各个组件间传递的 tuple 的字段名称已经事先定义好，所以 tuple 中只要按序填入各个 value 就行了，所以就是一个 value list.
5. Stream：源源不断传递的 tuple 就组成了 stream。stream grouping：即消息的 partition 方法。Storm 中提供若干种实用的 grouping 方式，包括 shuffle, fields hash, all, global, none, direct 和 localOrShuffle 等

相比于 s4, puma 等其他实时计算系统，storm 最大的亮点在于其记录级容错和能够保证消息精确处理的事务功能。下面就重点来看一下这两个亮点的实现原理。

- Storm 记录级容错的基本原理

首先来看一下什么叫做记录级容错？storm 允许用户在 spout 中发射一个新的源 tuple 时为其指定一个 message id, 这个 message id 可以是任意的 object 对象。多个源 tuple 可以共用一个 message id，表示这多个源 tuple 对用户来说是同一个消息单元。storm 中记录级容错的意思是说，storm 会告知用户每一个消息单元是否在指定时间内被完全处理了。那什么叫 做完全处理呢，就是该 message id 绑定的源 tuple 及由该源 tuple 后续生成的 tuple 经过了 topology 中每一个应该到达的 bolt 的处理。举个例子。在图 4-1 中，在 spout 由 message 1 绑定的 tuple1 和 tuple2 经过了 bolt1 和 bolt2 的处理生成两个新的 tuple，并最终都流向了 bolt3。当这个过程完成处理完时，称 message 1 被完全处理了。

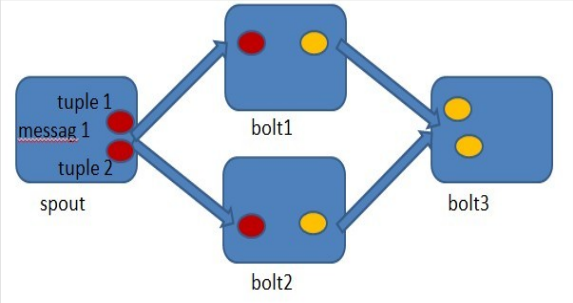

图 4-1

在 storm 的 topology 中有一个系统级组件，叫做 acker。这个 acker 的任务就是追踪从 spout 中流出来的每一个 message id 绑定的若干 tuple 的处理路径，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 就会告知 spout 该消息处理失败 了，相反则会告知 spout 该消息处理成功了。在刚才的描述中，我们提到了”记录 tuple 的处理路径”，如果曾经尝试过这么做的同学可以仔细地思考一下 这件事的复杂程度。但是 storm 中却是使用了一种非常巧妙的方法做到了。在说明这个方法之前，我们来复习一个数学定理。

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。

storm 中使用的巧妙方法就是基于这个定理。具体过程是这样的：在 spout 中系统会为用户指定的 message id 生成一个对应的 64 位整数，作为一个 root id。root id 会传递给 acker 及后续的 bolt 作为该消息单元的唯一标识。同时无论是 spout 还是 bolt 每次新生成一个 tuple 的时候，都会赋予该 tuple 一个 64 位的整数的 id。Spout 发射完某个 message id 对应的源 tuple 之后，会告知 acker 自己发射的 root id 及生成的那些源 tuple 的 id。而 bolt 呢，每次接受到一个输入 tuple 处理完之后，也会告知 acker 自己处理的输入 tuple 的 id 及新生 成的那些 tuple 的 id。Acker 只需要对这些 id 做一个简单的异或运算，就能判断出该 root id 对应的消息单元是否处理完成了。下面通过一个图示来说明这个过程。

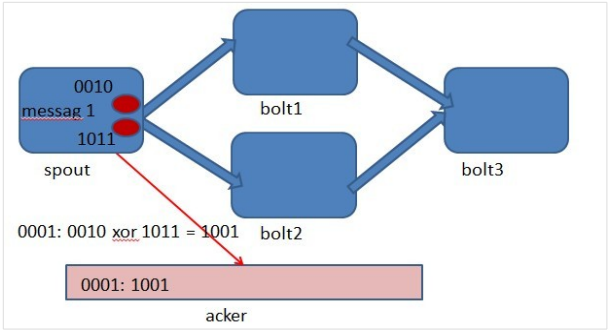

图 4-1 spout 中绑定 message 1 生成了两个源 tuple，id 分别是 0010 和 1011.

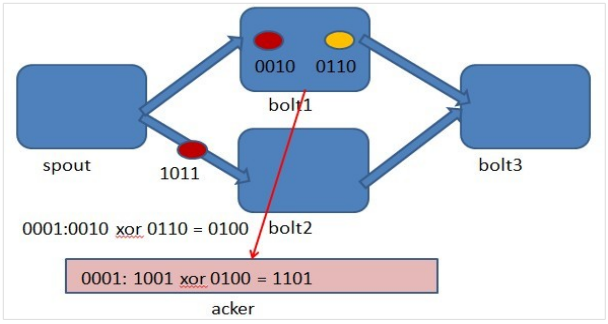

图 4-2 bolt1 处理 tuple 0010 时生成了一个新的 tuple，id 为 0110.

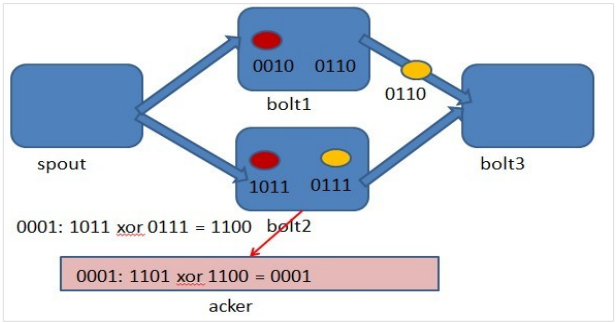

图 4-3 bolt2 处理 tuple 1011 时生成了一个新的 tuple，id 为 0111.

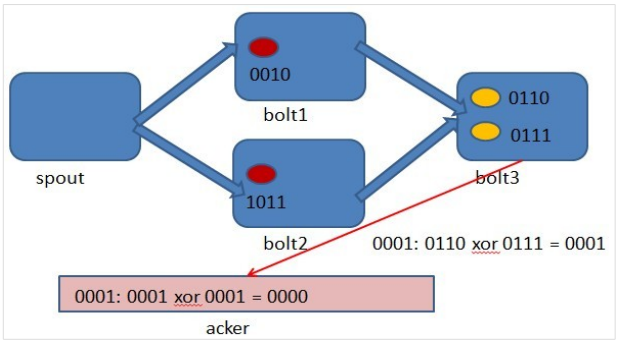

图 4-4 bolt3 中接收到 tuple 0110 和 tuple 0111，没有生成新的 tuple.

可能有些细心的同学会发现，容错过程存在一个可能出错的地方，那就是，如果生成的 tuple id 并不是完全各异的，acker 可能会在消息单元完全处理完成之前就错误的计算为 0。这个错误在理论上的确是存在的，但是在实际中其概率是极低极低的，完全可以忽略。

- Storm 的事务拓扑

事务拓扑(transactional topology)是 storm0.7 引入的特性，在最近发布的 0.8 版本中已经被封装为 Trident，提供了更加便利和直观的接口。因为篇幅所限，在此对事务拓扑做一个简单的介绍。

事务拓扑的目的是为了满足对消息处理有着极其严格要求的场景，例如实时计算某个用户的

成交笔数，要求结果完全精确，不能多也不能少。Storm 的事务拓扑 是完全基于它底层的 spout/bolt/acker 原语实现的，通过一层巧妙的封装得出一个优雅的实现。个人觉得这也是 storm 最大的魅力之一。

事务拓扑简单来说就是将消息分为一个个的批(batch)，同一批内的消息以及批与批之间的消息可以并行处理，另一方面，用户可以设置某些 bolt 为 committer，storm 可以保证 committer 的 finishBatch()操作是按严格不降序的顺序执行的。用户可以利用这个特性通过简单 的编程技巧实现消息处理的精确。

- Storm 在淘宝

由于 storm 的内核是 clojure 编写的(不过大部分的拓展工作都是 java 编写的)，为我们理解它的实现带来了一定的困难，好在大部分情况下 storm 都比较稳定，当然我们也在尽力熟悉 clojure 的世界。我们在使用 storm 时通常都是选择 java 语言开发应用程序。

在淘宝，storm 被广泛用来进行实时日志处理，出现在实时统计、实时风控、实时推荐等场景中。一般来说，我们从类 kafka 的 metaQ 或者基于 hbase 的 timetunnel 中读取实时日志消息，经过一系列处理，最终将处理结果写入到一个分布式存储中，提供给应用程序访问。我们每天的实时消息 量从几百万到几十亿不等，数据总量达到 TB 级。对于我们来说，storm 往往会配合分布式存储服务一起使用。在我们正在进行的个性化搜索实时分析项目中， 就使用了 timetunnel + hbase + storm + ups 的架构，每天处理几十亿的用户日志信息，从用户行为发生到完成分析延迟在秒级。

- Storm 的未来

Storm0.8 系列的版本已经在各大公司得到了广泛使用，最近发布的 1.0 版本中引入了多用户调度，安全机制等，关键词分组，支持各种数据库连接等等，使得其从一个纯计算框架演变成了一个包含存储和计算的实时计算新利器，还有刚才提到的 Trident，提供更加友好的接口，同时可定制 scheduler 的特性也为其针对不同的应用场景做优化提供了更便利的手段，也有人已经在基于 storm 的实时 ql(query language)上迈出了脚本。在服务化方面，storm 一直在朝着融入 mesos 框架的方向努力。同时，storm 也在实现细节上不断地优化，使用很多优秀的开源产品，包括 kryo, Disruptor, curator 等等。可以想象，当 storm 发展到 1.2 版本时，一定是一款无比杰出的产品，让我们拭目以待。