

Storm 基础

大数据计算主要有批量计算和流式计算两种形态，批量计算一般使用 Hadoop 做运算，主要优点是可以分布式大批量处理数据，但是缺点也很明显延迟大，处理速度慢。流失计算的出现缓解了 Hadoop 的缺陷。

目标

在本章中，您将学习：

- 实时计算概念
- Storm 安装



NIFI / training.com-china



实时计算概念

为什么实时？

实时系统是指该系统处于实时状态，响应应保证在规定的时间限制内或系统应满足规定的期限。大数据处理以离线批处理方式处理海量数据集。当对最新的数据集执行实时流处理时，我们操作的维度是现在或最近的过去；例如飞行控制系统、实时监控、信用卡欺诈检测、安全等等。延迟是这些分析中的一个关键方面。

响应速度和延迟正是 Hadoop 和相关的分布式批处理系统的不足之处。批处理系统的延迟无法控制在纳秒/毫秒内。在需要在几秒钟内得到准确结果的用例中，我们需要一个复杂的事件处理引擎来快速处理和派生结果。

互联网领域的实时计算一般都是针对海量数据进行的，除了像非实时计算的需求（如计算结果准确）以外，实时计算最重要的一个需求是能够实时响应计算结果，一般要求为秒级。互联网行业的实时计算可以分为以下两种应用场景：

数据源是实时的不间断的，要求对用户的响应时间也是实时的。

主要用于互联网流式数据处理。所谓流式数据是指将数据看作是数据流的形式来处理。数据流则是在时间分布和数量上无限的一系列数据记录的集合体；数据记录是数据流的最小组成单元。举个例子，对于大型网站，活跃的流式数据非常常见，这些数据包括网站的访问 PV/UV、用户访问了什么内容，搜索了什么内容等。实时的数据计算和分析可以动态实时地刷新用户访问数据，展示网站实时流量的变化情况，分析每天各小时的流量和用户分布情况，这对于大型网站来说具有重要的实际意义。

数据量大且无法或没必要预算，但要求对用户的响应时间是实时的。

主要用于特定场合下的数据分析处理。当数据量很大，同时发现无法穷举所有可能条件的查询组合或者大量穷举出来的条件组合无用的时候，实时计算就可以发挥作用，将计算过程推迟到查询阶段进行，但需要为用户提供实时响应[参考链接]。

批处理与实时处理

批处理

- 批处理访问所有数据。
- 它可以计算一些大而复杂的东西。
- 通常，它非常关注吞吐量。而不是计算中单个组件的延迟。
- 批处理的延迟以分钟或更长的时间计算。

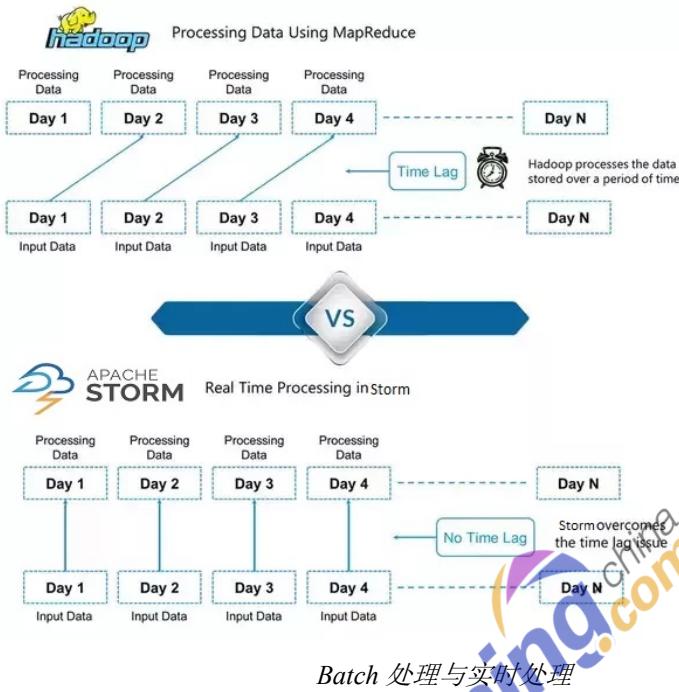
实时处理

- 实时处理有助于计算一个数据元素的函数。而且，可以说它计算了最近数据的一个小窗口。

- 实时处理计算的东西相对简单。
- 虽然我们需要接近实时的计算，最多只有几秒钟，但我们采用的是实时处理。
- 在实时处理中，计算通常是独立的。
- 它们本质上是异步的。这意味着数据源不直接与 stream（流）处理交互。

让我们比较 Hadoop 与 Storm 的例子

Hadoop	Apache Strom
它用于数据的分布式批处理。	它用于分布式实时数据处理。
由于数据的 Batch 处理，延迟很高。	由于数据的实时处理，延迟较低。
它本质上是有状态的，因此如果流停止，就需要保存最新的状态。	它是无状态的，因此实现起来更简单。
它的体系结构包括用于数据存储的 HDFS 和用于数据存储的 Map-Reduce。	它的结构由 spout 和 bolt 组成。
主从架构，不管有没有基于 zookeeper 的协调。主节点称为作业跟踪器，从节点称为任务跟踪器。	主从架构与基于 zookeeper 的协调。主节点称为 nimbus，从节点称为监督者。
MapReduce 作业按时间顺序执行并最终完成。	Apache Storm topology 一直运行到用户关闭或出现不可恢复的意外故障为止。
如果 JobTracker 挂掉，则会丢失所有活动的或正在运行的作业。	如果 nimbus/监督者挂掉，重新启动会使它从停止的地方继续，因此不会有任何改变或丢失。
两者都是开源、分布式和容错的	



实时计算相关技术

互联网上海量数据（一般为日志流）的实时计算过程可以被划分为以下三个阶段：数据的产生与收集阶段、传输与分析处理阶段、存储对对外提供服务阶段。下面分别进行简单的介绍：



■ 数据实时采集

需求：功能上保证可以完整的收集到所有日志数据，为实时应用提供实时数据；响应时间上要保证实时性、低延迟在 1 秒左右；配置简单，部署容易；系统稳定可靠等。

目前，互联网企业的海量数据采集工具，有 Facebook 开源的 Scribe、LinkedIn 开源的 Kafka、Cloudera 开源的 Flume，淘宝开源的 TimeTunnel、Hadoop 的 Chukwa 等，均可以满足每秒数百 MB 的日志数据采集和传输需求。

■ 数据实时计算

传统的数据操作，首先将数据采集并存储在 DBMS 中，然后通过 query 和 DBMS 进行交互，得到用户想要的答案。整个过程中，用户是主动的，而 DBMS 系统是被动的。



但是，对于现在大量存在的实时数据，比如股票交易的数据，这类数据实时性强，数据量大，没有止境，传统的架构并不合适。流计算就是专门针对这种数据类型准备的。在流数据不断变化的运动过程中实时地进行分析，捕捉到可能对用户有用的信息，并把结果发送出去。整个过程中，数据分析处理系统是主动的，而用户却是处于被动接收的状态。



实时计算的步骤

■ 实时查询服务

全内存：直接提供数据读取服务，定期 dump 到磁盘或数据库进行持久化。

半内存：使用 Redis、Memcache、MongoDB、BerkeleyDB 等内存数据库提供数据实时查询服务，由这些系统进行持久化操作。

全磁盘：使用 HBase 等以分布式文件系统（HDFS）为基础的 NoSQL 数据库，对于 key-value 引擎，关键是设计好 key 的分布。

实时计算主流产品

■ Yahoo 的 S4

S4 是一个通用的、分布式的、可扩展的、分区容错的、可插拔的流式系统，Yahoo 开发 S4 系统，主要是为了解决：搜索广告的展现、处理用户的点击反馈。

■ Twitter 的 Storm

是一个分布式的、容错的实时计算系统。可用于处理消息和更新数据库(流处理)，在数据流上进行持续查询，并以流的形式返回结果到客户端(持续计算)，并行化一个类似实时查询的热点查询(分布式的 RPC)。

■ Facebook 的 Puma

Facebook 使用 puma 和 HBase 相结合来处理实时数据，另外 Facebook 发表一篇利用 HBase/Hadoop 进行实时数据处理的论文(ApacheHadoop Goes Realtime at Facebook)，通过一些实时性改造，让批处理计算平台也具备实时计算的能力。

■ NaviSite

是使用 Storm 作为事件日志监视/审计系统。系统中生成的每个日志都将经历 storm。Storm 将根据配置的正则表达式对消息进行检查，如果有匹配的消息，那么该特定消息将保存到数据库中。

■ Wego

是一个位于新加坡的旅游元搜索引擎。与旅游相关的数据来自世界各地，时间不同。Storm 帮助 Wego 搜索实时数据，解决并发问题，并为最终用户找到最佳匹配。

应用举例

对于电子商务网站上的店铺：

1. 实时展示一个店铺的到访顾客流水信息，包括访问时间、访客姓名、访客地理位置、访客 IP、访客正在访问的页面等信息；
2. 显示某个到访顾客的所有历史来访记录，同时实时跟踪显示某个访客在一个店铺正在访问的页面等信息；
3. 支持根据访客地理位置、访问页面、访问时间等多种维度下的实时查询与分析。
4. 集成消息队列来执行数据清洗，消息首先进入消息队列（例如 kafka），storm 将数据发射到集群里面，通过 bolt 来执行清洗，并存入数据库中。

总结

1. 并不是任何应用都做到实时计算才是最好的。
2. 使用哪些技术和框架来搭建实时计算系统，需要根据实际业务需求进行选择。
3. 对于分布式系统来说，系统的可配置性、可维护性、可扩展性十分重要，系统调优永无止境。

Storm 集群搭建

Storm 集群部署需要按以下步骤依次安装

1. 搭建 Zookeeper 集群;
2. 安装 Storm 依赖库;
3. 下载并解压 Storm 发布版本;
4. 修改 storm.yaml 配置文件;
5. 启动 Storm 各个后台进程。

说明：zookeeper 集群安装参考前面介绍的 zookeeper 安装部署

安装 Storm 依赖库

Storm 的依赖库需要安装在 Nimbus 和管理机器上，包括以下组件：

- Java
- Python 2.6.6

假设上面的组件已经放在 /opt/niit/tools 目录中

说明：JDK 安装在讲授 Hadoop 时已经学习过了。在 CentOS 7 中，默认情况下已有 python 依赖关系

安装 Python2.6.6

- 下载 Python2.6.6
wget <http://www.python.org/ftp/python/2.6.6/Python-2.6.6.tar.bz2>
- 编译安装 Python2.6.6
tar -jxvf Python-2.6.6.tar.bz2
cd Python-2.6.6
.configure
make
make install
- 测试 Python2.6.6
\$ python -V
Python 2.6.6

配置 Storm

Storm 配置

```
cd /opt/niit/tools  
unzip storm-0.8.2.zip
```

Storm 发行版本解压目录下有一个 conf/storm.yaml 文件，用于配置 Storm，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"
```

- storm.zookeeper.port: zookeeper 的端口

- 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）的加载路径，默认情况下为 “/usr/local/lib:/opt/local/lib:/usr/lib”，一般来说 ZMQ 和 JZMQ 本来就是默认安装在 /usr/local/lib 下，因此不需要配置。

```
java.library.path: "/usr/local/lib:/opt/local/lib:/usr/lib"
```

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 jar 文件、conf 文件等，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
```

Storm 启动

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：

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 连接。

为了方便使用，可以将 bin/storm 加入到系统环境变量中。

storm.yaml 默认配置

```
##### These all have default values as shown
##### Additional configuration goes into storm.yaml

java.library.path: "/usr/local/lib:/opt/local/lib:/usr/lib"

### storm.* configs are general configurations
# the local dir is where jars are kept
storm.local.dir: "storm-local"
storm.zookeeper.servers:
  - "localhost"
storm.zookeeper.port: 2181
storm.zookeeper.root: "/storm"
storm.zookeeper.session.timeout: 20000
storm.zookeeper.connection.timeout: 15000
storm.zookeeper.retry.times: 5
storm.zookeeper.retry.interval: 1000
storm.zookeeper.retry.intervalceiling.millis: 30000
storm.cluster.mode: "distributed" # can be distributed or local
storm.local.mode.zmq: false
storm.thrift.transport: "backtype.storm.security.auth.SimpleTransportPlugin"
storm.messaging.transport: "backtype.storm.messaging.netty.Context"

### nimbus.* configs are for the master
nimbus.host: "localhost"
nimbus.thrift.port: 6627
nimbus.thrift.max_buffer_size: 1048576
```

```
nimbus.childopts: "-Xmx1024m"
nimbus.task.timeout.secs: 30
nimbus.supervisor.timeout.secs: 60
nimbus.monitor.freq.secs: 10
nimbus.cleanup.inbox.freq.secs: 600
nimbus.inbox.jar.expiration.secs: 3600
nimbus.task.launch.secs: 120
nimbus.reassign: true
nimbus.file.copy.expiration.secs: 600
nimbus.topology.validator: "backtype.storm.nimbus.DefaultTopologyValidator"

### ui.* configs are for the master
ui.port: 8080
ui.childopts: "-Xmx768m"

logviewer.port: 8000
logviewer.childopts: "-Xmx128m"
logviewer.appender.name: "A1"

drpc.port: 3772
drpc.worker.threads: 64
drpc.queue.size: 128
drpc.invocations.port: 3773
drpc.request.timeout.secs: 600
drpc.childopts: "-Xmx768m"

transactional.zookeeper.root: "/transactional"
transactional.zookeeper.servers: null
transactional.zookeeper.port: null

### supervisor.* configs are for node supervisors
# Define the amount of workers that can be run on this machine. Each worker is
assigned a port to use for communication
supervisor.slots.ports:
  - 6700
  - 6701
  - 6702
  - 6703
supervisor.childopts: "-Xmx256m"
#how long supervisor will wait to ensure that a worker process is started
supervisor.worker.start.timeout.secs: 120
#how long between heartbeats until supervisor considers that worker dead and
tries to restart it
supervisor.worker.timeout.secs: 30
#how frequently the supervisor checks on the status of the processes it's
monitoring and restarts if necessary
supervisor.monitor.frequency.secs: 3
#how frequently the supervisor heartbeats to the cluster state (for nimbus)
supervisor.heartbeat.frequency.secs: 5
supervisor.enable: true

### worker.* configs are for task workers
worker.childopts: "-Xmx768m"
worker.heartbeat.frequency.secs: 1
```

```
task.heartbeat.frequency.secs: 3
task.refresh.poll.secs: 10

zmq.threads: 1
zmq.linger.millis: 5000
zmq.hwm: 0

storm.messaging.netty.server_worker_threads: 1
storm.messaging.netty.client_worker_threads: 1
storm.messaging.netty.buffer_size: 5242880 #5MB buffer
storm.messaging.netty.max_retries: 30
storm.messaging.netty.max_wait_ms: 1000
storm.messaging.netty.min_wait_ms: 100

### topology.* configs are for specific executing storms
topology.enable.message.timeouts: true
topology.debug: false
topology.optimize: true
topology.workers: 1
topology.acker.executors: null
topology.tasks: null
# maximum amount of time a message has to complete before it's considered failed
topology.message.timeout.secs: 30
topology.skip.missing.kryo.registrations: false
topology.max.task.parallelism: null
topology.max.spout.pending: null
topology.state.synchronization.timeout.secs: 60
topology.stats.sample.rate: 0.05
topology.builtin.metrics.bucket.size.secs: 60
topology.fall.back.on.java.serialization: true
topology.worker.childopts: null
topology.executor.receive.buffer.size: 1024 #batched
topology.executor.send.buffer.size: 1024 #individual messages
topology.receiver.buffer.size: 8 # setting it too high causes a lot of problems
(heartbeat thread gets starved, throughput plummets)
topology.transfer.buffer.size: 1024 # batched
topology.tick.tuple.freq.secs: null
topology.worker.shared.thread.pool.size: 4
topology.disruptor.wait.strategy: "com.lmax.disruptor.BlockingWaitStrategy"
topology.spout.wait.strategy: "backtype.storm.spout.SleepSpoutWaitStrategy"
topology.sleep.spout.wait.strategy.time.ms: 1
topology.error.throttle.interval.secs: 10
topology.max.error.report.per.interval: 5
topology.kryo.factory: "backtype.storm.serialization.DefaultKryoFactory"
topology.tuple.serializer:
"backtype.storm.serialization.types.ListDelegateSerializer"
topology.trident.batch.emit.interval.millis: 500
dev.zookeeper.path: "/tmp/dev-storm-zookeeper"
```



活动 1.1: **Strom** 安装

NIIT | training.com-china

练习问题

1. storm 安装部署依赖以下哪些 ()
 - a. zeroMQ
 - b. JZMQ
 - c. zookeeper
 - d. hdfs
2. 互联网海量数据的实时计算过程可以被划分为除以下哪个阶段外
 - a. 数据实时采集
 - b. 数据实时计算
 - c. 实时查询服务
 - d. 数据归档
3. 下面哪项不是 storm 的命令 ()
 - a. bin/storm nimbus >/dev/null 2>&1 &
 - b. bin/storm supervisor >/dev/null 2>&1 &
 - c. bin/storm ui >/dev/null 2>&1 &
 - d. bin/storm work >/dev/null 2>&1 &
4. 下列哪项不是 storm 的特征?
 - a. 容错
 - b. 语言不可知论
 - c. 保证消息的传递
 - d. 低延迟
5. 在 storm.yaml 中以下哪个属性没有设置下列?
 - a. storm.zookeeper.servers
 - b. Storm.zookeeper.port
 - c. storm.local.dir
 - d. storm.zookeeper.ip

小结

在本章中，您已学习了流式计算的相关知识：

- 实时计算概念
- Batch 处理与实时处理
- 实时计算相关技术
- 实时计算主流产品
- 应用举例
- Storm 的依赖库需要安装在 Nimbus 和管理机器上，包括以下组件：
 - Python 2.6.6
 - Java
- Storm 安装
- 配置 Storm.yaml

通过安装 Storm 让您对 Storm 的组件有个初步的了解，下一章节将介绍 Storm 的相关组件以及如何构建一个 topology

Storm 组件

本章中将重点讲述 Storm 的特性基本概念，讲述 topology、tuple、spout、bolt 的基本概念以及如何创建一个 Topology。

Storm 集群的表面上类似于 Hadoop 集群。Hadoop 上运行的是“MapReduce 作业”，Storm 上运行的是“topology”。

“作业”和“topology”本身非常不同一个关键的区别为 MapReduce 作业最终会完成，而 topology 会永远处理消息（或者直到杀死它的进程）。

目标

在本章中，您将学习：

- Storm 基本概念
- 构建 Topology



Storm 概念

Storm 出现之前，你可能需要自己手动维护一个由消息队列和消息处理器所组成的实时处理网络，消息处理器从消息队列取出一个消息进行处理，更新数据库，发送消息给其它队列，等等等。不幸的是，这种方式有以下几个缺陷：

- 单调乏味：你花费了绝大部分开发时间去配置把消息发送到哪里，部署消息处理器，部署中间消息节点 — 你的大部分时间花在设计，配置这个数据处理框架上，而你真正关心的消息处理逻辑在你的代码里面占的比例很少。
- 脆弱：不够健壮，你要自己写代码保证所有的消息处理器和消息队列正常运行。
- 伸缩性差：当一个消息处理器的消息量达到阀值，你需要对这些数据进行分流，你需要配置这些新的消息处理器以让他们处理分流的消息。

Storm 特性

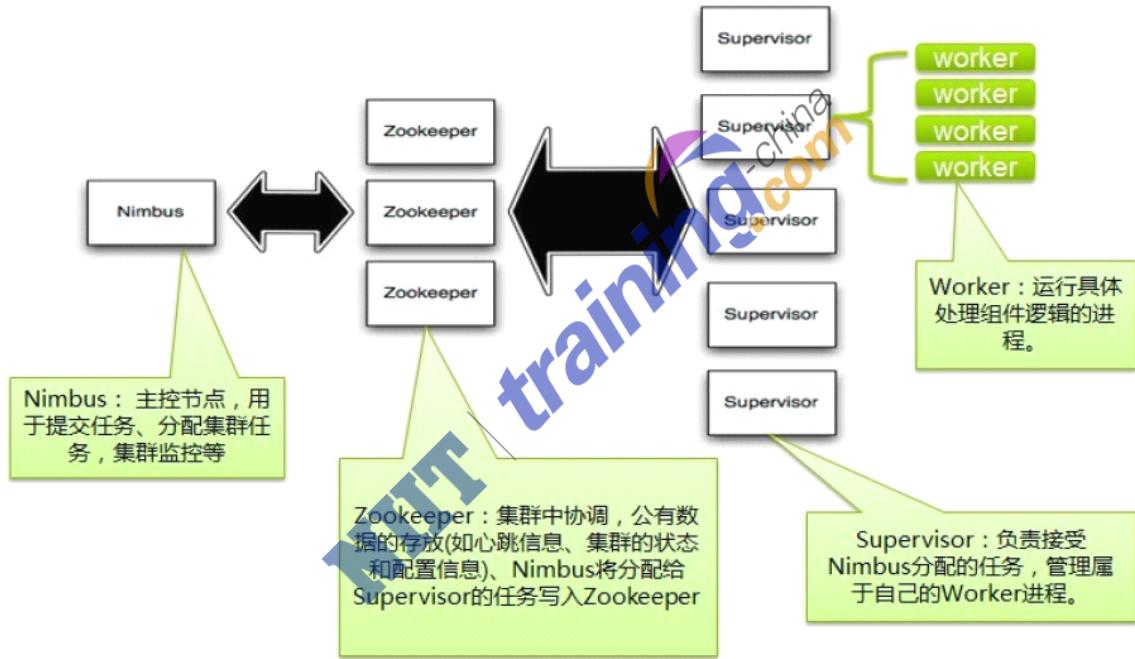
Storm 定义了一批实时计算的原语。如同 hadoop 大大简化了并行批量数据处理，storm 的这些原语大大简化了并行实时数据处理。storm 的一些关键特性如下：

- **适用场景广泛：** storm 可以用来处理消息和更新数据库（消息流处理），对一个数据量进行持续的查询并返回客户端（持续计算），对一个耗资源的查询作实时并行化的处理（分布式方法调用）， storm 的这些基础原语可以满足大量的场景。
- **可伸缩性高：** Storm 的可伸缩性可以让 storm 每秒可以处理的消息量达到很高。为了扩展一个实时计算任务，你所需要做的就是增加机器并且提高这个计算任务的并行度设置（parallelism setting）。作为 Storm 可伸缩性的一个例证，一个 Storm 应用可在在一个 10 个节点的集群上每秒处理 1000000 个消息 — 包括每秒一百多次的数据库调用。Storm 使用 ZooKeeper 来协调集群内的各种配置使得 Storm 的集群可以很容易的扩展。
- **保证无数据丢失：** 实时系统必须保证所有的数据被成功的处理。那些会丢失数据的系统的适用场景非常窄，而 storm 保证每一条消息都会被处理，这一点和 S4 相比有巨大的反差。
- **异常健壮：** 不像 Hadoop — 出了名的难管理， storm 集群非常容易管理。容易管理是 storm 的设计目标之一。
- **容错性好：** 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处理逻辑。 storm 保证一个处理逻辑永远运行 — 除非你显式杀掉这个处理逻辑。
- **语言无关性：** 健壮性和可伸缩性不应该局限于一个平台。Storm 的 topology 和消息处理组件可以用任何语言来定义，这一点使得任何人都可以使用 storm。

Storm 基本组件

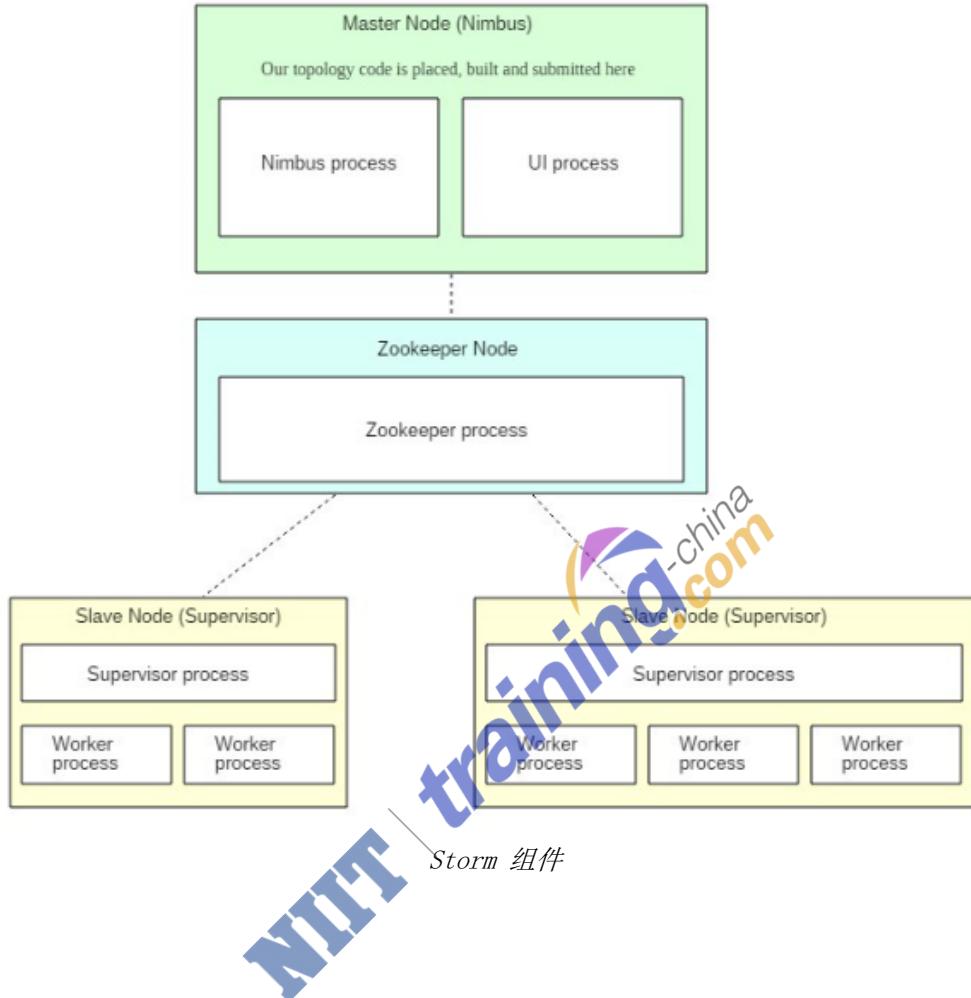
在 Storm 的集群里面有两种节点： 控制节点(master node)和工作节点(worker node)。控制节点上面运行一个后台程序： Nimbus。Nimbus 负责在集群里面分布代码，分配工作给机器，并且监控状态。

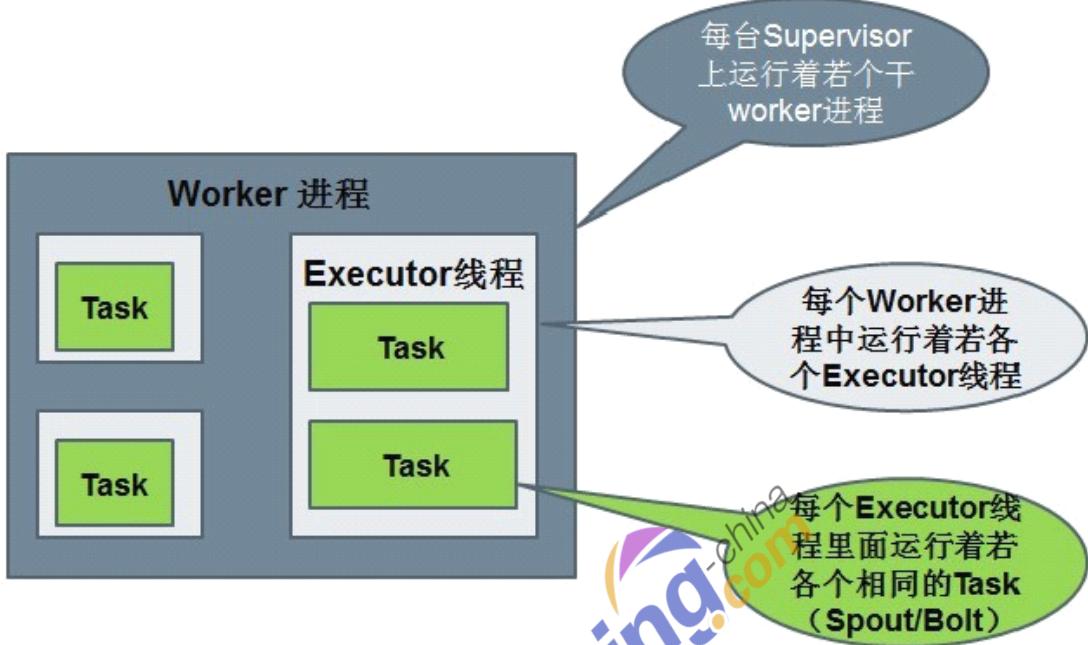
每一个工作节点上面运行一个叫做 Supervisor 的节点。Supervisor 会监听分配给它那台机器的工作，根据需要启动/关闭工作进程。每个 supervisor 上运行着若干个 worker 进程（根据配置文件 supervisor.slots.ports 进行配置）。在一个 worker 进程中，又包含多个 Executor 线程，一个 Executor 线程中，可以包含一个或多个相同的 Task (spout/bolt)，默认一个 Executor 线程包含一个 task。每一个工作进程执行一个 Topology 的一个子集；一个运行的 Topology 由运行在很多机器上的很多工作进程组成。



storm topology 结构

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。





Worker 进程示意图

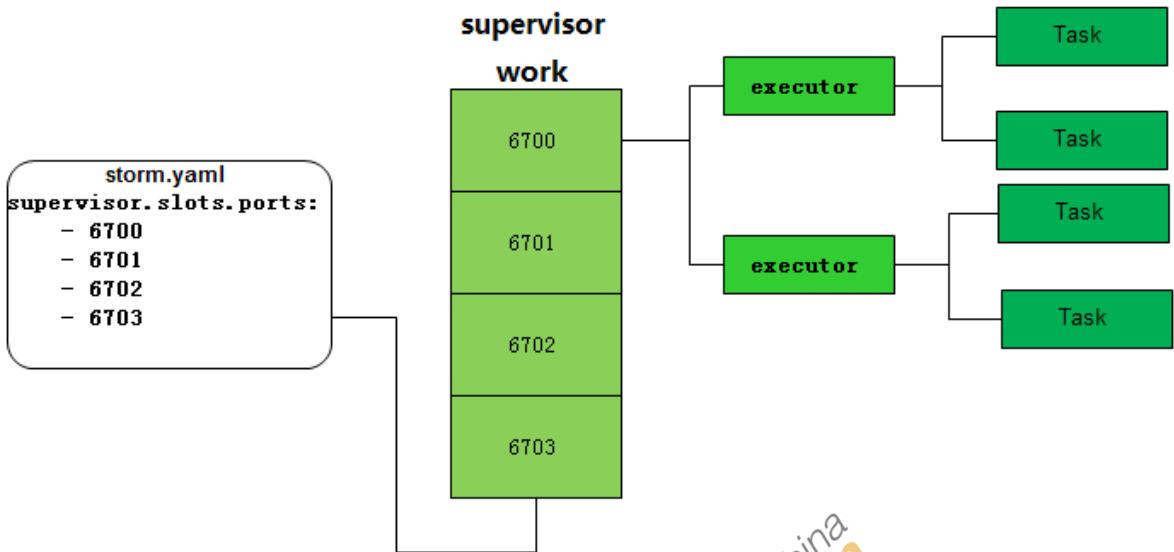
下图描述了一个这样的示例：

一台 supervisor 上配置了 4 个 work
程序提交了的并行度是这样：

```
topologyBuilder.setBolt("boltTest", new BoltTest(), 2).setNumTasks(4)  
.shuffleGrouping("green-bolt");
```

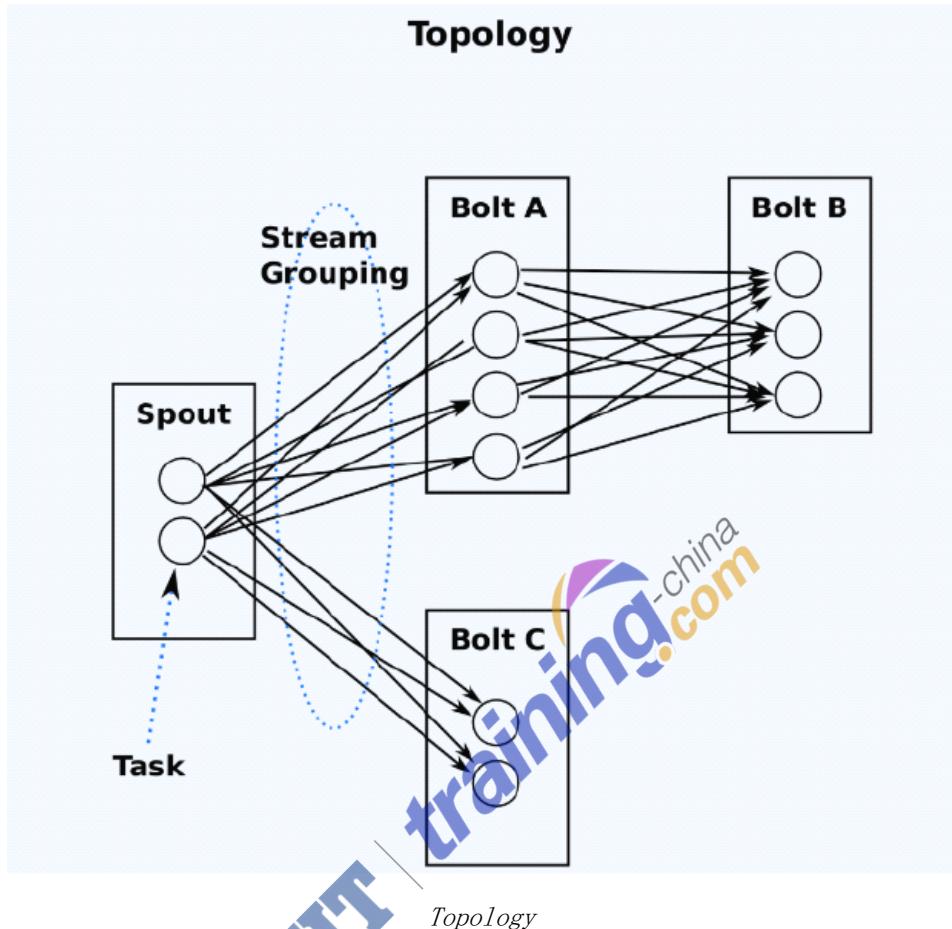
这个代码含义如下：

程序产生了 2 个线程 executor
每个线程运行了 2 个 Task



关键组件

- **Topology (拓扑)**：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。Topology 是一组由 Spouts (数据源) 和 Bolts (数据操作) 通过 Stream Groupings 进行连接组成的图。



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所有任务会被转移到其他机器上。以下代码定义一个 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"));
```

- **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。
- **Stream:** 消息流 stream 是 storm 里的关键抽象，以 tuple 为单位组成的一条有向无界的 data 流。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。

```

/**
 * flow definition method
 */
public void declareOutputFields(OutputFieldsDeclarer declarer) {
    //information flow definition for the default ID
    declarer.declare(new Fields("word", "count"));
    //customize the message flow definition for ID
    declarer.declareStream("streamId", new Fields("word", "count"));
}

```

- **Spout:** 从来源处读取数据并放入 topology。Spout 分成可靠和不可靠两种；当 Storm 接收失败时，可靠的 Spout 会对 tuple（元组，数据项组成的列表）进行重发；而不可靠的 Spout 不会考虑接收成功与否只发射一次。而 Spout 中最主要的方法就是 nextTuple()，该方法会发射一个新的 tuple 到 topology，如果没有新 tuple 发射则会简单的返回。消息源可以发射多条消息流 stream。用 OutputFieldsDeclarer.declareStream 来定义多个 stream，然后使用 SpoutOutputCollector 来发射指定的 stream。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。

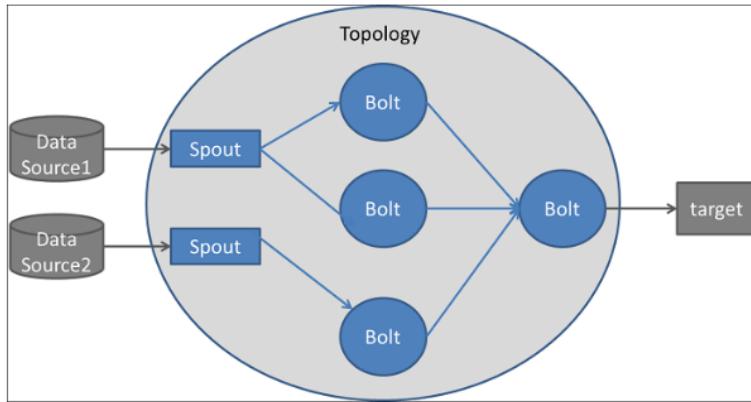
spout 中两个比较重要的方法 ack 和 fail。storm 在检测到一个 tuple 被整个 topology 成功处理的时候调用 ack，否则调用 fail。storm 只对可靠的 spout 调用 ack 和 fail。

```

/**
 * Two message flows are defined
 */
public void declareOutputFields(OutputFieldsDeclarer declarer){
    declarer.declareStream("streamId1", new Fields("word1"));
    declarer.declareStream("streamId2", new Fields("word2"));
}
/**
 * Emit corresponding messages based on the message flow
 */
public void nextTuple(){
    collector.emit("streamId1", new Values (streamId1's word1));
    collector.emit("streamId2", new Values (streamId2's word2));
}

```

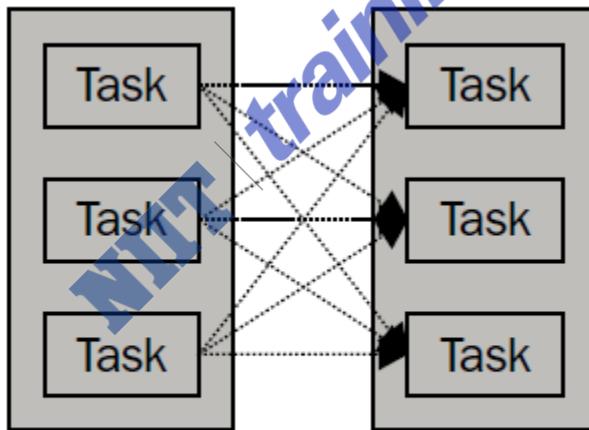
- **Bolt:** Topology 中所有的处理都由 Bolt 完成。Bolt 可以完成任何事，比如：连接的过滤、聚合、访问文件、数据库等等。Bolt 从 Spout 中接收数据并进行处理，如果遇到复杂流的处理也可能将 tuple 发送给另一个 Bolt 进行处理。而 Bolt 中最重要的方法是 execute()，以新的 tuple 作为参数接收。不管是 Spout 还是 Bolt，如果将 tuple 发射成多个流，这些流都可以通过 declareStream 来声明。
- Bolts 和 spouts 一样，可以发射多条消息流，使用 OutputFieldsDeclarer.declareStream 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。



Spout 和 bolt 的组合实例

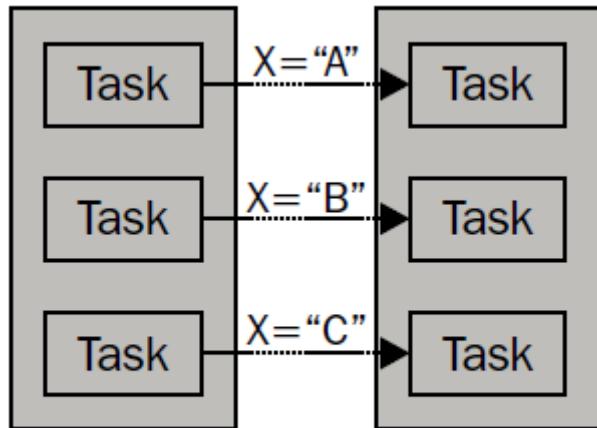
- **Stream Groupings（流分组）**：Stream Grouping 定义了一个流在 Bolt 任务间该如何被切分，谁来处理哪些数据，按照什么规则来分配。
这里有 Storm 提供的 6 个 Stream Grouping 类型：

- 随机分组（Shuffle grouping）：随机分发 tuple 到 Bolt 的任务，保证每个任务获得相等数量的 tuple。当不需要任何数据驱动的分区时，这种分组是理想的。



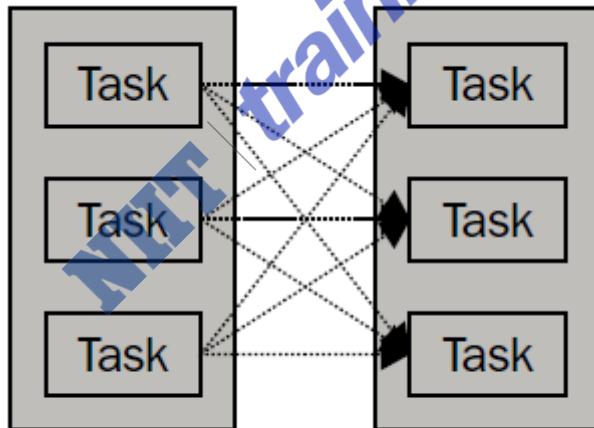
Shuffle Grouping

- 字段分组（Fields grouping）：根据指定字段分割数据流，并分组。例如，根据“user-id”字段，相同“user-id”的元组总是分发到同一个任务，不同“user-id”的元组可能分发到不同的任务。



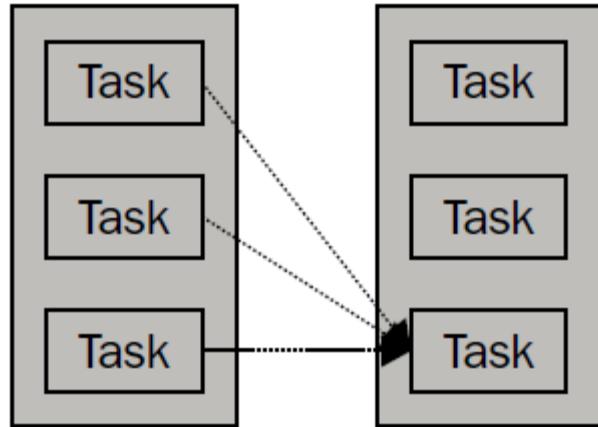
Fields Grouping

- 全部分组 (All grouping) : tuple 被复制到 bolt 的所有任务。这种类型需要谨慎使用。例如，如果你在对流进行某种过滤，那么你必须将过滤参数传递给所有的 bolt。这可以通过将这些参数发送到一个流中来实现，该流是由所有 bolt 任务与所有分组共同订阅的。



All Grouping

- 全局分组 (Global grouping) : 全部流都分配到 bolt 的同一个任务。明确地说，是分配给 ID 最小的那个 task。这种情况的一般用例是，在你的 topology 中需要一个 reduce 阶段，而在这个阶段中，你希望将 topology 中以前步骤的结果组合在一个单独的螺栓中。



Global Grouping

- 无分组 (None grouping)：你不需要关心流是如何分组。目前，无分组等效于随机分组。但最终，Storm 将把无分组的 Bolts 放到 Bolts 或 Spouts 订阅它们的同一线程去执行（如果可能）。
- 直接分组 (Direct grouping)：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。消息处理器可以通过 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)。
- **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。
- **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）。

```
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。

- **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）。
- **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）。



构建 Topology

实现的目标：

我们将设计一个 topology，来实现对 hadoop 中单词数量统计的程序，目的是让大家对于 topology 快速上手，有一个初步的理解。

设计 Topology 结构：

在开始开发 Storm 项目的第一步，就是要设计 topology。确定数据处理逻辑，



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

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：

- public class WordReader extends BaseRichSpout 用来读取一个文件，并将一段话发射到 bolt
- public class WordSpliter extends BaseBasicBolt 获取一句话，并做分词拆分，将拆分后的单词发送到其他 Bolt
- public class WordCounter extends BaseBasicBolt 统计每个单词的数量
- public class WordCountTopo 构建一个 topology 来运行该程序

编写代码

■ Spout 要求

Spout 是流的源头，负责读或者监听外部的消息源。需要继承

backtype.storm.topology.IRichSpout.BaseRichSpout 接口，spout 有如下方法需要实现

```
public void open(Map conf, TopologyContext context, SpoutOutputCollector collector)
```

集群中的一个 Work 在初始化这个组件的时候调用 open 方法，这个方法只会被调用一次。

```
void close();
```

组件关闭的时候执行，例如执行了 kill -9

```
void fail(Object msgId);
```

当给定 msgid 的消息处理失败或者超时的时候会被调用，程序应该做一些操作以便在 nextTuple 中被调用

```
void ack(Object msgId);
```

当一个 tuple 被处理成功的时候会调用此方法，这个时候程序应该标记这个 tuple 已经被处理了，然后需要做一些必要的清理，比如从消息队列中移走这个 tuple。

```
void nextTuple();
```

这个方法被 storm 调用来处理消息源中的下一个数据，在这个方法内部需要读取消息源的下一个数据并通过 backtype.storm.spout.ISpoutOutputCollector 来发射这个消息。

例如 Spout 定义格式如下：

```
public class RandomSentenceSpout extends BaseRichSpout {  
    SpoutOutputCollector _collector;  
    Random _rand;  
  
    @Override  
    public void open(Map conf, TopologyContext context,  
    SpoutOutputCollector collector) {  
        _collector = collector;  
        _rand = new Random();  
    }  
  
    @Override  
    public void nextTuple() {  
        Utils.sleep(100);  
        String[] sentences = new String[]{"the cow jumped over the moon",  
        "an apple a day keeps the doctor away",  
        "four score and seven years ago", "snow white and the seven  
        dwarfs", "i am at two with nature"};  
        String sentence = sentences[_rand.nextInt(sentences.length)];  
        _collector.emit(new Values(sentence));  
    }  
  
    @Override  
    public void ack(Object id) {  
    }  
  
    @Override  
    public void fail(Object id) {  
    }  
  
    @Override  
    public void declareOutputFields(OutputFieldsDeclarer declarer) {  
        declarer.declare(new Fields("word"));  
    }  
}
```

■ Bolt 要求

Topology 中流的具体处理组件，需要继承 backtype.storm.topology.IBasicBolt 类。

```
void prepare(Map stormConf, TopologyContext context);
```

和 spout 的方法性质一样，初始化 bolt 信息，只执行一次。

```
public void execute(Tuple input, BasicOutputCollector collector)
```

该方法执行来之 bolt 订阅的输入流中的每个消息，然后根据 public void

declareOutputFields(OutputFieldsDeclarer declarer) 定义的格式发射到下一个 bolt 中

```
public static class SplitSentence extends BaseBasicBolt {
```

```
    public SplitSentence() {
```

```
}
```

```
@Override
```

```
public void declareOutputFields(OutputFieldsDeclarer declarer) {
```

```
    declarer.declare(new Fields("word"));
```

```
}
```

```
@Override
```

```
public void execute(Tuple input, BasicOutputCollector collector) {
```

```
    // TODO Auto-generated method stub
```

```
    String ary_tuple[] = input.getString(0).split(" "));
```

```
    for (String value : ary_tuple)
```

```
        collector.emit(new Values(value));
```

```
}
```

```
}
```

■ 执行方式：

Storm 分本地模式和远程模式这 2 种运行模式来运行一个 topology。

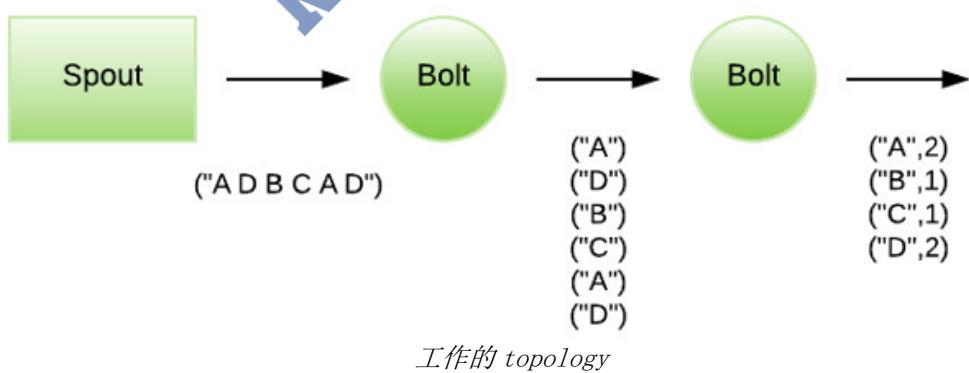
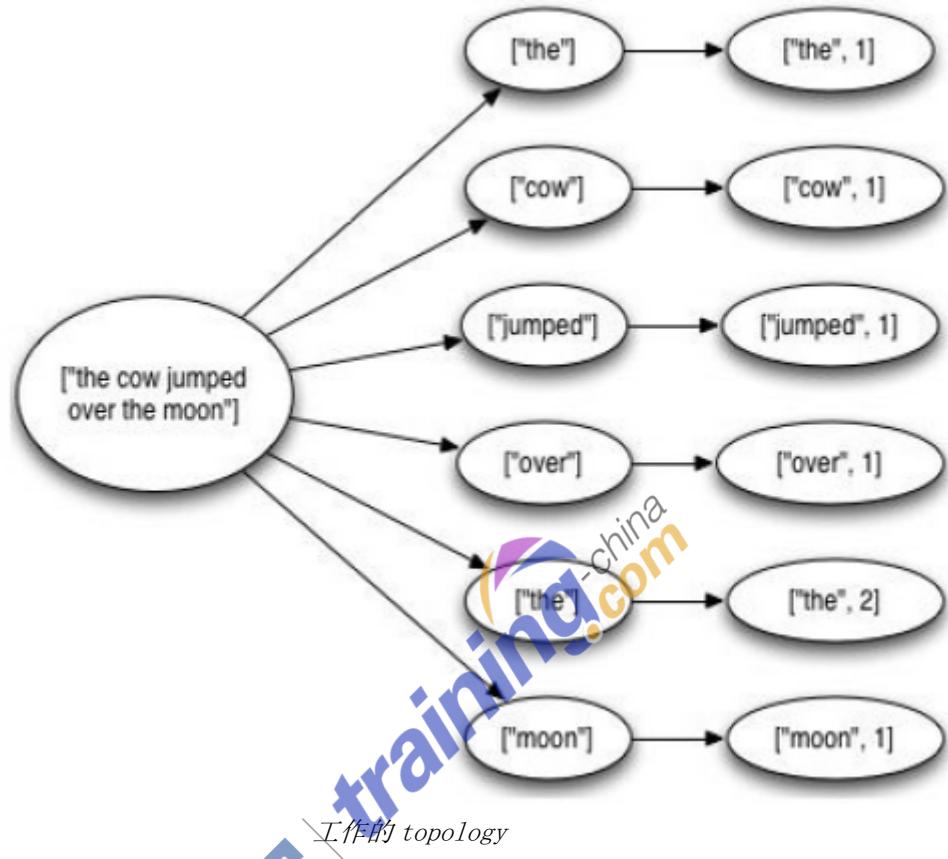
本地模式： storm topology 运行在本地的单个 JVM 中，主要是用来测试和 debug

```
LocalCluster cluster = new LocalCluster();
```

```
cluster.submitTopology("word-count", conf, builder.createTopology());
```

远程模式：通过 storm client 提交 topology 到集群

```
StormSubmitter.submitTopology("first", conf, builder.createTopology());
```





活动 2.1：使用 **IDEA** 创建一个基于 **Maven** 的项目



活动 2.2：通过流式计算，构建一个 **topology** 并统计一段话中每个单词的数量



活动 2.3：将 **topology** 程序打包上传到服务器运行

NIIT | training.com-china

练习问题

1. 下列哪些不是 storm 的特性 ()
 - a. 使用场景广泛，可伸缩性高
 - b. 异常健壮且容错性好
 - c. 只能用 java 语言调用 storm
 - d. 保证数据无丢失
2. 关于 strom 组成中，哪项不是 Storm 的组件
 - a. Nimbus
 - b. Jobtracker
 - c. Supervisor
 - d. Topology
3. 关于 storm 中流分组的说啊，哪项不是 storm 的流分组
 - a. Shuffle grouping
 - b. Fields grouping
 - c. Global grouping
 - d. Type grouping
4. 哪一种随机分配元组到 Bolt 的任务会保证每个分组的元组数目相等?
 - a. Shuffle grouping
 - b. Fields grouping
 - c. Global grouping
 - d. All grouping
5. Storm Topology 本地运行在单个 JVM 中是哪一种运行模式?
 - a. Local mode
 - b. Remote mode

小结

在本章中，您学习了：

- Storm 基本概念
 - 适用场景广泛
 - 可伸缩性高
 - 保证无数据丢失
 - 异常强大
 - 良好的容错性
 - 语言无关性
- Strom 基本组件
 - Nimbus
 - Supervisor
 - Zookeeper
- Strom 其他组件
 - Topology
 - Tuple
 - Workers
 - Tasks（任务）
 - Configuration（配置）
 - Stream Grouping
 - Streams（流）
 - Reliability（可靠性）
- 构建 Topology
 - 定义 Spout : Spout 是流的源头，负责读或者监听外部的消息源
 - 定义 Bolt : Topology 中所有处理都是由 Bolt 完成的。
 - 主程序连接 spout 和 bolt 与 topology
- 通过使用 storm 来实时计算一段话中每个单词数量的 topology，初步体验了如何创建一个 topology 以及各组件的工作原理。

深入讲解 Storm

Storm 可以实时分布式持续计算，如何提升分布式计算的效率则是本章的内容描述重点，通过配置 Storm 的并行度来加速 Storm 并行处理的能力，本章将讲述 Storm 关于并行度的概念以及配置。同时还需要了解 Storm 的容错机制

目标

在本章中，您将学习：

- 并行度概念
- 消息的可靠处理
- Storm 失败任务处理
- Storm 调度器



并行度概念

并行度总览

storm 的并行指的是由非常多的 supervisor 组成。

storm 的 supervisor 运行的是 topology 中的 task(spout/bolt)。

task 是 storm 中进行计算的最小的运行单位，表示的是 spout、bolt 的运行实例。

程序执行的最大粒度的运行单位是进程。在 supervisor 中，运行 task 的进程称作 worker。

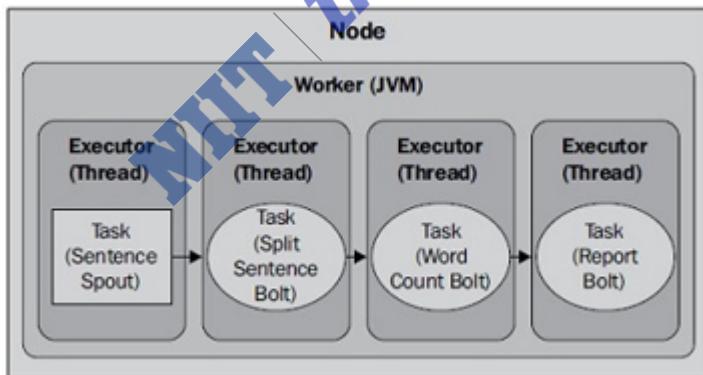
supervisor 节点上可以运行非常多的 worker。

在 worker 中可以运行线程的，这些线程称作 executor。在 executor 中，运行 task。

总结一下，supervisor(节点)>worker(进程)>executor(线程)>task(实例)

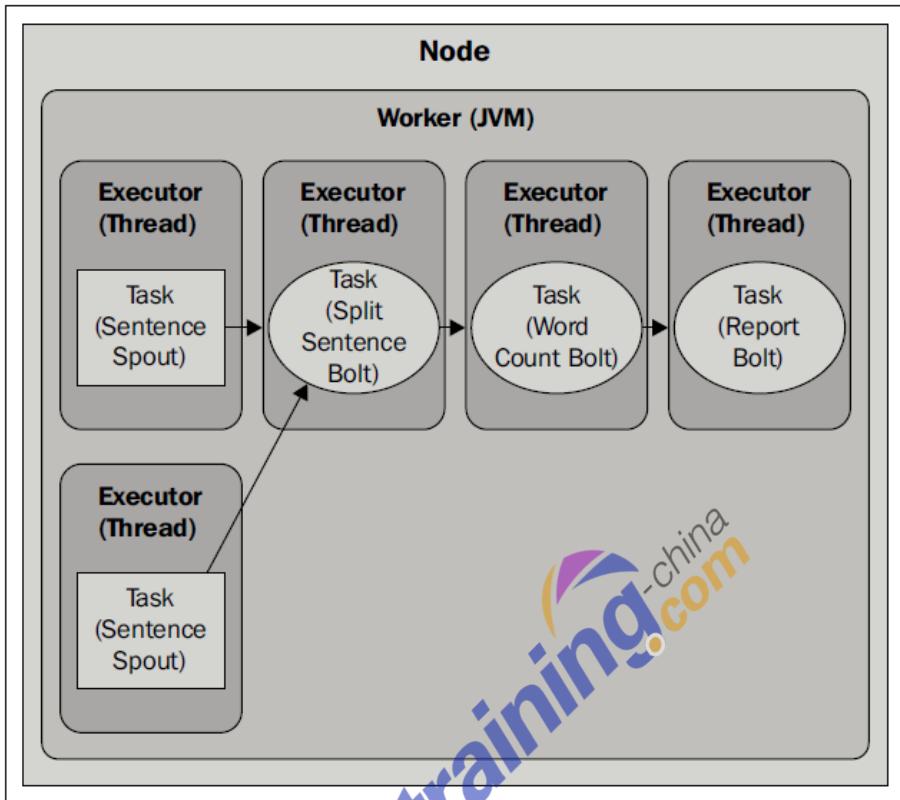
- Worker 过程 (JVMs) :这些是运行在节点上的独立 JVM 进程。每个节点被配置为运行一个或多个 worker。一个 topology 可以请求分配一个或多个 worker。
- Executors(threads): 这些是在工作 JVM 进程中运行的 Java 线程。多个任务可以分配给一个 executor。默认情况下，Storm 将为每个 executor 分配一个 task。
- Task (bolt/spout 实例) : Task 是 spout 和 bolts 的实例，executor 线程调用它们的 nextTuple()和 execute()方法。

Storm 为 topology 中定义的每个组件创建 1 个 task，并为每个 task 分配 1 个 executor。Storm 的并行度 API 允许你设置每个 spout/bolt 的 executor 数量以及总的 task 数量，从而对这种行为进行控制。



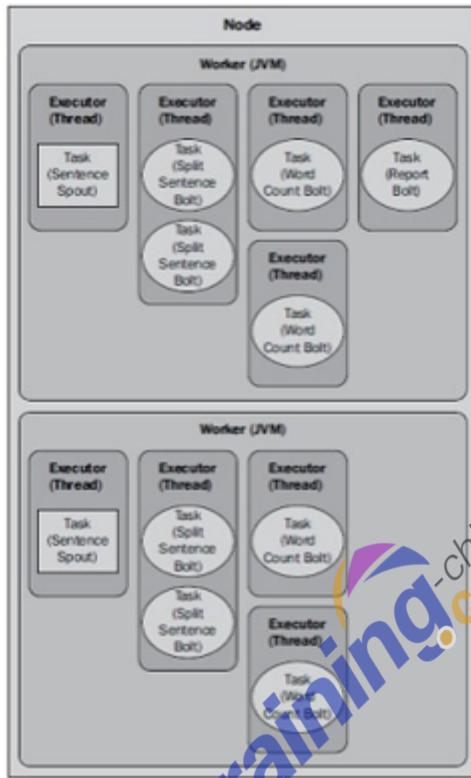
默认单个 worker, executor 和 task

现在如果我们为 SentenceSpout 配置任务数为 2。



各组件的关系

接下来，我们将设置 split sentence bolt 作为四个 task，两个 executor 执行。每个 executor 线程将被分配 2 个 task 来执行($4 / 2 = 2$)。我们还将配置 word count bolt 作为 4 个 task 运行，每个任务都有自己的 executor 线程，还将添加 1 个 worker。

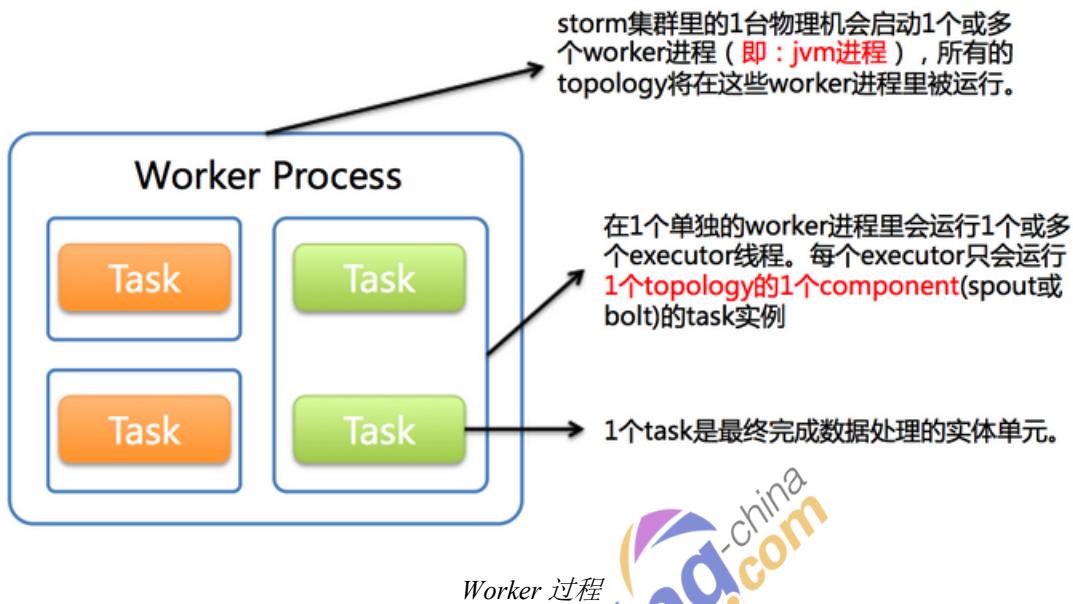


2 个 worker, 多个 executor 和多个 task

必须指出的是，在本地模式下运行 topology 时，增加 worker 的数量没有任何影响。在本地模式下运行的 topology 总是在单个 JVM 进程中运行，因此只有任务和执行器并行性设置有影响。Storm 的本地模式为集群行为提供了很好的近似，对于开发非常有用，但是在转移到生产环境之前，你应该始终在真正的集群环境中测试你的应用程序。

注释

Storm 当前的配置设置优先级如下：defaults.yaml, storm.yaml, 特定于 topology 的配置，特定于内部组件的配置，特定于外部组件的配置



Storm 集群中的其中 1 台机器可能运行着属于多个拓扑(可能为 1 个)的多个 worker 进程(可能为 1 个)。每个 worker 进程运行着特定的某个拓扑的 executors。

1 个或多个 executor 可能运行于 1 个单独的 worker 进程, 每 1 个 executor 从属于 1 个被 worker process 生成的线程中。每 1 个 executor 运行着相同的组件(spout 或 bolt)的 1 个或多个 task。

1 个 task 执行着实际的数据处理(实际数据处理器)。

1 个 worker 进程执行一个拓扑的子集。1 个 worker 进程从属于 1 个特定的拓扑, 并运行着这个拓扑的 1 个或多个组件(spout 或 bolt)的 1 个或多个 executor。一个运行中的拓扑包括集群中的许多台机器上的许多个这样的进程。

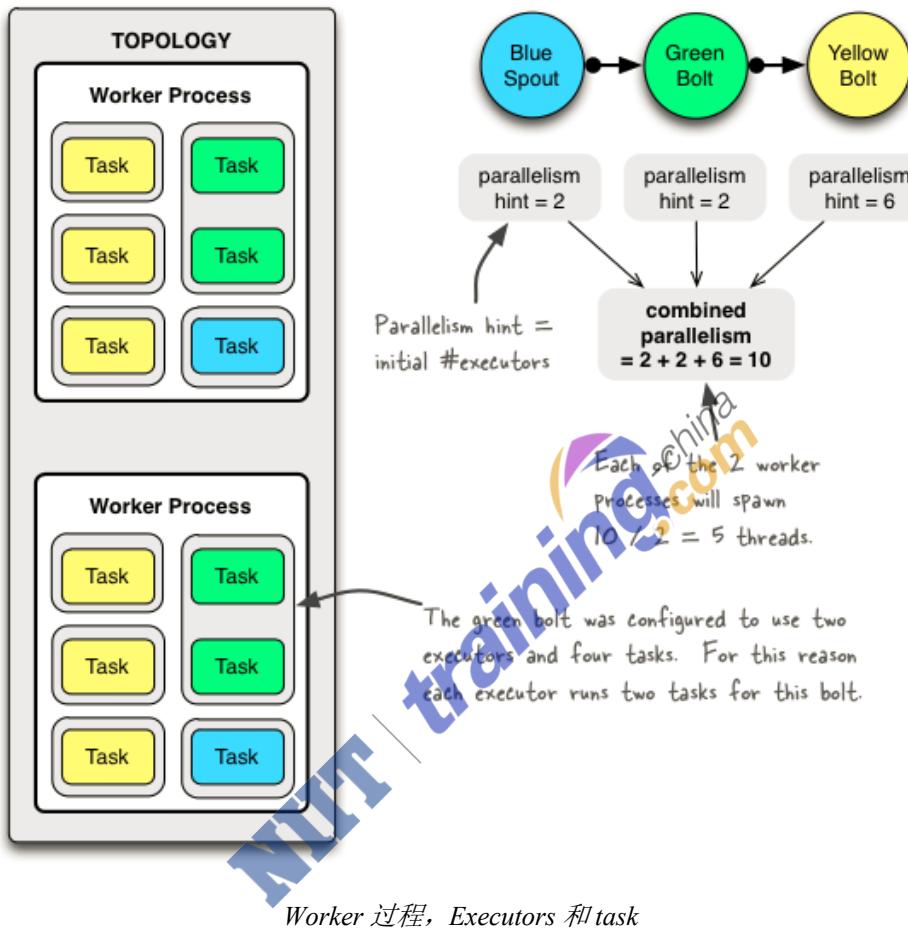
1 个 executor 是 1 个 worker 进程生成的 1 个线程。它可能运行着 1 个相同的组件(spout 或 bolt)的 1 个或多个 task。

1 个 task 执行着实际的数据处理, 你用代码实现的每一个 spout 或 bolt 就相当于分布于整个集群中的许多个 task。在 1 个拓扑的生命周期中, 1 个组件的 task 的数量总是一样的, 但是 1 个组件的 executor(线程)的数量可以随着时间而改变。这意味着下面的条件总是成立: thread 的数量 \leq task 的数量。默认情况下, task 的数量与 executor 的数量一样, 例如, Storm 会在每 1 个线程运行 1 个 task。

配置拓扑的并发度

Storm 的术语“并发度(parallelism)”是特别用来描述所谓的 parallelism hint 的, 这代表 1 个组件的初始的 executor(线程)的数量。在此文档中我们使用术语“并发度”的一般意义来描述的是, 你不但可以配置 executor 的数量, 还可以配置 worker 进程的数量, 还可以是 1 个拓扑的 task 的数量。

下面的图表展示了 1 个简单拓扑在实际操作中看起来是怎样的。这个拓扑包含了 3 个组件：1 个 spout 叫做 BlueSpout，2 个 bolt 分别叫 GreenBolt 和 YellowBolt。BlueSpout 发送它的输出到 GreenBolt，GreenBolt 又把它的输出发到 YellowBolt。



3 个组件的并发度加起来是 10，就是说拓扑一共有 10 个 executor，一共有 2 个 worker，每个 worker 产生 $10 / 2 = 5$ 条线程。

Greem bolt 配置成 2 个 executor 和 4 个 task。为此每个 executor 为这个 bolt 运行 2 个 task。

下面的代码配置了这 3 个组件，相关代码如下：

```
Config conf = new Config();
conf.setNumWorkers(2); // 使用 2 个 worker 进程
topologyBuilder.setSpout("blue-spout", new BlueSpout(), 2); //parallelism hint 为 2
topologyBuilder.setBolt("green-bolt", new GreenBolt(),
2) .setNumTasks(4) .shuffleGrouping("blue-spout");
topologyBuilder.setBolt("yellow-bolt", new YellowBolt(),
6) .shuffleGrouping("green-bolt");
```

```
StormSubmitter.submitTopology( "mytopology", conf,  
topologyBuilder.createTopology() );
```

wordcountTopology 的代码解释在前一节里

```
Config config = new Config();  
config.setNumWorkers(2);  
builder.setSpout(SENTENCE_SPOUT_ID, spout, 2);  
builder.setBolt(SPLIT_BOLT_ID, splitBolt, 2).setNumTasks(4)  
.shuffleGrouping(SENTENCE_SPOUT_ID);  
builder.setBolt(COUNT_BOLT_ID, countBolt, 4)  
.fieldsGrouping(SPLIT_BOLT_ID, new Fields("word"));
```

更新运行中的 Topology 的并行度

Storm 可以在不需要重启集群或拓扑，来增加或减少 worker 进程和 executor 的数量。这样方式叫 rebalancing。

使用命令行工具来做：

重新配置拓扑 “mytopology” 使用 5 个 worker 进程。

spout “blue-spout” 使用 3 个 executor

bolt “yellow-bolt” 使用 10 个 executor

```
$ storm rebalance mytopology -n 5 -e blue-spout=3 -e yellow-bolt=10
```

注意点

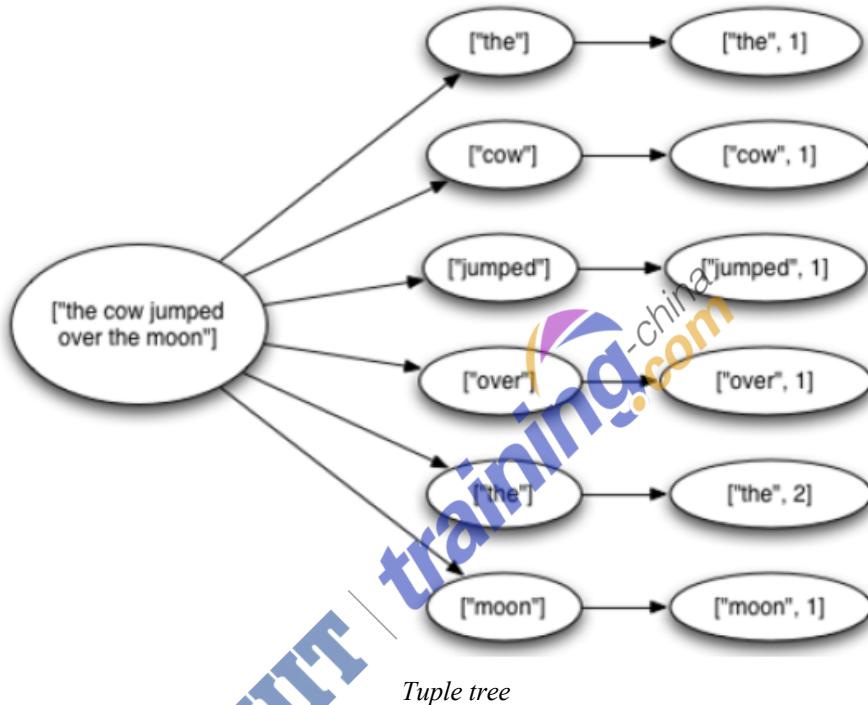
executor 的数目

- executor 是真正的并行度（事实上的并行度）。(task 数目是想要设置的并行度)
- executor 初始数目=spout 数目+bolt 数目+acker 数目（这些加起来也就是 task 数目。）
- spout 数目，bolt 数目，acker 数目运行时是不会变化的，但是 executor 数目可以变化。
- TASK 的存在只是为了 topology 扩展的灵活性，与并行度无关

消息可靠性处理

消息处理结束的定义

一个消息(tuple)从 spout 发送出来，可能会被处理并产生更多的消息。这些消息构成一个树状结构，我们称之为“tuple tree”，例如下图：



spout 认为消息已经被处理结束有两种情况，一个是 tuple tree 不再生长，另一个树中的所有消息被标识为“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。

消息的生命周期

■ Spout 方法的调用顺序为：

- declareOutputFields()
- open()
- activate()
- nextTuple() (loop call)
- deactivate()

■ 流程如下：

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 会为这个消息提供一个 message ID，它将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")。

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。

说明：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 spout 任务来应答。

可靠性 API

STORM 可靠性需要调用如下 2 个方法：

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。

为 tuple tree 中指定的节点增加一个新的节点，例如在 bolt 中调用 collector.emit(tuple, new Values(word))；我们称之为锚定（anchoring）。锚定是在我们发送消息的同时进行的。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。

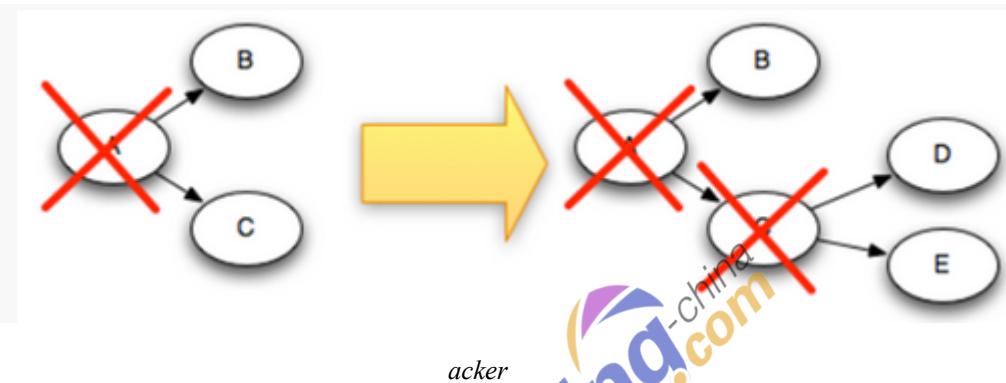
acker

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 任务的并行度。

为了理解 Storm 可靠性处理机制，我们从研究一个消息的生命周期和 tuple tree 的管理入手。当一个消息被创建的时候（无论是在 spout 还是 bolt 中），系统都为该消息分配一个 64bit 的随机值作为 id。这些随机的 id 是 acker 用来跟踪由 spout 消息派生出来的 tuple tree 的。

每个消息都知道它所在的 tuple tree 对应的根消息的 id。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。



关于 Storm 如何跟踪 tuple tree，我们再深入的探讨一下。前面说过系统中可以有任意个数的 acker，那么，每当一个消息被创建或应答的时候，它怎么知道应该通知哪个 acker 呢？

系统使用一种哈希算法来根据 spout 消息的 messageId 确定由哪个 acker 跟踪此消息派生出来的 tuple tree。因为每个消息都知道与之对应的根消息的 messageId，因此它知道应该与哪个 acker 通信。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。

tuple 是如何被跟踪的呢？系统中有成千上万的消息，如果为每个 spout 发送的消息都构建一棵树的话，很快内存就会耗尽。所以，必须采用不同的策略来跟踪每个消息。由于使用了新的跟踪算法，Storm 只需要固定的内存（大约 20 字节）就可以跟踪一棵树。这个算法是 storm 正确运行的核心，也是 storm 最大的突破。

acker 任务保存了 spout 消息 id 到一对值的映射。第一个值就是 spout 的任务 id，通过这个 id，acker 就知道消息处理完成时该通知哪个 spout 任务。第二个值是一个 64bit 的数字，我们称之为“ack val”，它是树中所有消息的随机 id 的异或结果。ack val 表示了整棵树的状态，无论这棵树多大，只需要这个固定大小的数字就可以跟踪整棵树。当消息被创建和被应答的时候都会有相同的消息 id 发送过来做异或运算。

每当 acker 发现一棵树的 ack val 值为 0 的时候，它就知道这棵树已经被完全处理了。因为消息的随机 ID 是一个 64bit 的值，因此 ack val 在树处理完之前被置为 0 的概率非常小。假设你每秒钟发送一万个消息，从概率上说，至少需要 50,000,000 年才会有机会发生一次错误。即使如此，也只有在这个消息确实处理失败的情况下才会有数据的丢失。

总结：

有三种方法可以控制消息的可靠处理机制：

将变量 Config.TOPOLOGY_ACKERS 设置为 0，或设置 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(l));

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(l), 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));

Spout 可靠性处理代码如下：

```
collector.emit(newValues("field1","field2",3),msgId);
```

Bolt 方法的可靠性处理代码如下：

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

Storm 任务失败处理

任务级别失败

- **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。

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。

Storm 调度器

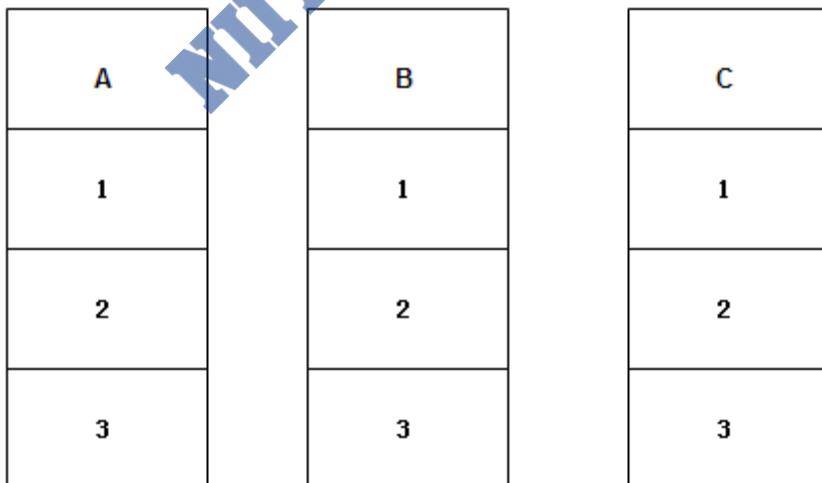
讨论调度器之前回顾下几个基本概念：

- Slot。这代表一个 Supervisor 节点上的一个单位资源。每个 slot 对应一个 port，一个 slot 只能被一个 Worker 占用。
- Worker, Executor, Task, 1 个 Worker 包含 1 个或多个 Executor 执行器，每个执行器包含多个 Task。
- Executor 的表现形式为[1-1]、[2-2]，中括号内的数字代表该 Executor 中的起始 Task id 到末尾 Task id，1 个 Worker 就相当于在外面加个大括号{[1-1],[2-2]}
- Component。Storm 中的每个组件就是指一个 Spout 类型或一个 Bolt 类型，这里指的是名称类型，不包含个数。

调度器

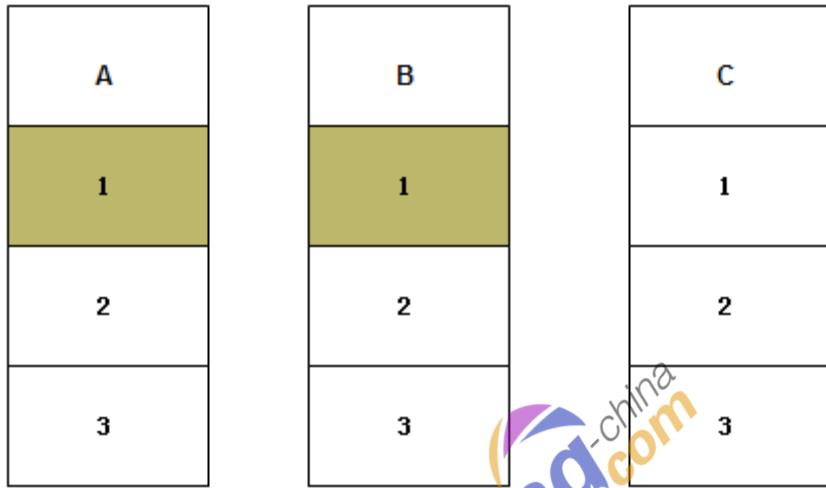
scheduler 是 storm 的调度器，它负责为 Topology 分配当前的集群可用资源，目前 storm 提供了 3 种调度器：

- EvenScheduler：会将系统中的资源均匀的分配给当前需要任务分配的多个 Topology。
- DefaultScheduler：跟 EvenScheduler 基本一致，只是分配前，先释放其他 Topology 不需要的资源，然后调用 EvenScheduler。
- IsolationScheduler：可以单独为某些 Topology 指定它们需要的机器资源。以下介绍 EvenScheduler 调度器的原理。假设集群中有 3 台机器 A、B、C，每台机器上有 3 个 port。如下示意图：



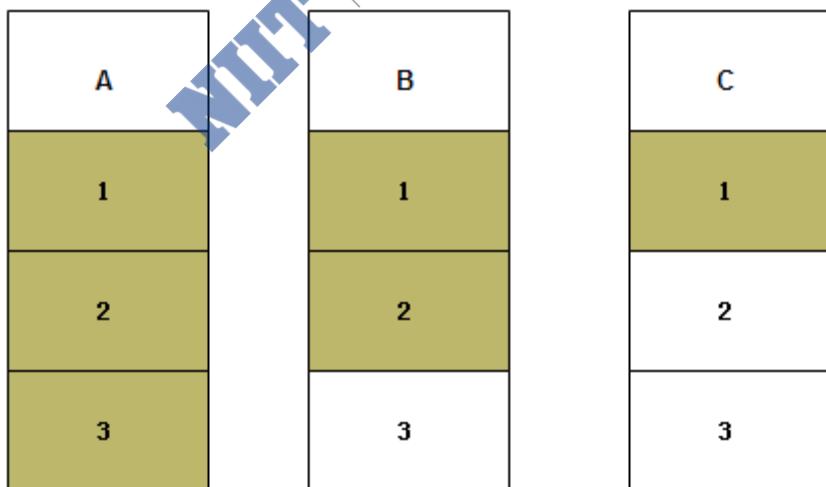
T1 和 T2 可用的资源

当有新的 Topology 提交的时候，就会分配资源，这时，在获取集群可用资源后会先 sort 一下，然后就会按顺序取 slot 分配给需要的 Topology。由此可见，3 台机器 sort 之后的排序为：[A, 1]、[B, 1]，[C, 1]，[A, 2]，[B, 2]，[C, 2]，[A, 3]，[B, 3]，[C, 3]，现在有两个 Topology，T1 和 T2，T1 需要 2 个 slot，T2 需要 4 个 slot。T1 提交时，会用掉前两个 slot [A, 1], [B, 1]，如下图：



为 T1 分配的资源

T2 提交时，这时集群可用资源为[A, 2], [B, 2], [C, 1], [A, 3], [B, 3], [C, 2], [C, 3]（排序之后的顺序），则 T2 会占用[A, 2], [B, 2], [C, 1], [A, 3]，如下图：



为 T1 和 T2 分配的资源

A 全部被占用，而 C 却只被占用了一个。由此可见，Storm 现有的 Scheduler 会导致集群资源分配不均匀情况，而且也没有考虑 worker 之间的通信负载情况。

调度器用法

在 storm.yaml 里面作如下配置

```
storm.scheduler: "backtype.storm.scheduler.EvenScheduler"
```

自定义调度器

如要需要自定义调度器，需要实现 IScheduler 接口

```
public interface IScheduler {  
  
    void prepare(Map conf);  
  
    /**  
     * Set assignments for the topologies which needs scheduling.  
     * The new assignments is available  
     * through <code>cluster.getAssignments()</code>  
     *  
     * @param topologies all the topologies in the cluster,  
     * some of them need schedule. Topologies object here  
     * only contain static information about topologies.  
     * Information like assignments, slots are all in  
     * the <code>cluster</code>object.  
     * @param cluster the cluster these topologies are running in.  
     * <code>cluster</code> contains everything user  
     * need to develop a new scheduling logic.  
     * e.g. supervisors information, available slots, current  
     * assignments for all the topologies etc.  
     * User can set the new assignment for topologies using  
     * <code>cluster.setAssignmentById</code>  
     */  
    void schedule(Topologies topologies, Cluster cluster);  
}
```

参数说明

Topologies 包含当前集群里面运行的所有 Topology 的信息： StormTopology 对象，配置信息，以及从 task 到组件(bolt, spout)id 的映射信息。

Cluster 对象则包含了当前集群的所有状态信息：对于系统所有 Topology 的 task 分配信息，所有的 supervisor 信息等等。

```
import backtype.storm.scheduler.*;  
import clojure.lang.PersistentArrayMap;  
import java.util.*;  
public class DirectScheduler implements IScheduler{  
    @Override  
    public void prepare(Map conf) {  
    }  
    @Override
```

```

public void schedule(Topologies topologies, Cluster cluster) {
    System.out.println("DirectScheduler: begin scheduling");
    // Gets the topology which we want to schedule
    Collection<TopologyDetails> topologyDetailes;
    TopologyDetails topology;
    //作业是否要指定分配的标识
    String assignedFlag;
    Map map;
    Iterator<String> iterator = null;
    topologyDetailes = topologies.getTopologies();
    for(TopologyDetails td: topologyDetailes){
        map = td.getConf();
        assignedFlag = (String)map.get("assigned_flag");
        //如何找到的拓扑逻辑的分配标为1则代表是要分配的,否则走系统的调度
        if(assignedFlag != null && assignedFlag.equals("1")){
            System.out.println("finding topology named " + td.getName());
            topologyAssign(cluster, td, map);
        }else {
            System.out.println("topology assigned is null");
        }
    }
    //其余的任务由系统自带的调度器执行
    new EvenScheduler().schedule(topologies, cluster);
}
/***
 * 拓扑逻辑的调度
 * @param cluster
 * 集群
 * @param topology
 * 具体要调度的拓扑逻辑
 * @param map
 * map 配置项
 */
private void topologyAssign(Cluster cluster, TopologyDetails topology, Map map) {
    Set<String> keys;
    PersistentArrayMap designMap;
    Iterator<String> iterator;
    iterator = null;
    // make sure the special topology is submitted,
    if (topology != null) {
        designMap = (PersistentArrayMap)map.get("design_map");
        if(designMap != null){
            System.out.println("design map size is " + designMap.size());
            keys = designMap.keySet();
            iterator = keys.iterator();
            System.out.println("keys size is " + keys.size());
        }
        if(designMap == null || designMap.size() == 0){
            System.out.println("design map is null");
        }
        boolean needsScheduling = cluster.needsScheduling(topology);
        if (!needsScheduling) {

```

```

        System.out.println("Our special topology does not need scheduling.");
    } else {
        System.out.println("Our special topology needs scheduling.");
        // find out all the needs-scheduling components of this topology
        Map<String, List<ExecutorDetails>> componentToExecutors =
cluster.getNeedsSchedulingComponentToExecutors(topology);
System.out.println("needs scheduling(component->executor): " +
componentToExecutors);
        System.out.println("needs scheduling(executor->components): " +
cluster.getNeedsSchedulingExecutorToComponents(topology));
SchedulerAssignment currentAssignment =
cluster.getAssignmentById(topology.getId());
        if (currentAssignment != null) {
System.out.println("current assignments: " +
currentAssignment.getExecutorToSlot());
        } else {
            System.out.println("current assignments: {}");
        }
        String componentName;
        String nodeName;
        if(designMap != null && iterator != null){
            while (iterator.hasNext()){
                componentName = iterator.next();
                nodeName = (String)designMap.get(componentName);
System.out.println("现在进行调度 组件名称->节点名称:" + componentName + "->" +
nodeName);
componentAssign(cluster, topology, componentToExecutors, componentName, nodeName);
            }
        }
    }
}

/**
 * 组件调度
 * @param cluster
 * 集群的信息
 * @param topology
 * 待调度的拓扑细节信息
 * @param totalExecutors
 * 组件的执行器
 * @param componentName
 * 组件的名称
 * @param supervisorName
 * 节点的名称
 */
private void componentAssign(Cluster cluster, TopologyDetails topology,
Map<String, List<ExecutorDetails>> totalExecutors, String componentName, String
supervisorName) {
    if (!totalExecutors.containsKey(componentName)) {
        System.out.println("Our special-spout does not need scheduling.");
    }
}

```

```

} else {
    System.out.println("Our special-spout needs scheduling.");
    List<ExecutorDetails> executors = totalExecutors.get(componentName);

    // find out the our "special-supervisor" from the supervisor metadata
    Collection<SupervisorDetails> supervisors = cluster.getSupervisors().values();
    SupervisorDetails specialSupervisor = null;
    for (SupervisorDetails supervisor : supervisors) {
        Map meta = (Map) supervisor.getSchedulerMeta();
        if (meta != null && meta.get("name") != null) {
            System.out.println("supervisor name:" + meta.get("name"));
            if (meta.get("name").equals(supervisorName)) {
                System.out.println("Supervisor finding");
                specialSupervisor = supervisor;
                break;
            }
        } else {
            System.out.println("Supervisor meta null");
        }
    }
    // found the special supervisor
    if (specialSupervisor != null) {
        System.out.println("Found the special-supervisor");
        List<WorkerSlot> availableSlots = cluster.getAvailableSlots(specialSupervisor);
        // 如果目标节点上已经没有空闲的 slot, 则进行强制释放
        if (availableSlots.isEmpty() && !executors.isEmpty()) {
            for (Integer port : cluster.getUsedPorts(specialSupervisor)) {
                cluster.freeSlot(new WorkerSlot(specialSupervisor.getId(), port));
            }
        }
        // 重新获取可用的 slot
        availableSlots = cluster.getAvailableSlots(specialSupervisor);
        // 选取节点上第一个 slot, 进行分配
        cluster.assign(availableSlots.get(0), topology.getId(), executors);
        System.out.println("We assigned executors:" + executors + " to slot: [" +
            " + availableSlots.get(0).getNodeId() + ", " + availableSlots.get(0).getPort() +
            "]");
    } else {
        System.out.println("There is no supervisor find!!!");
    }
}
}

```

主函数说明：

```

int numOfParallel;
TopologyBuilder builder;
StormTopology stormTopology;
Config config;
//待分配的组件名称与节点名称的映射关系
HashMap<String, String> component2Node;
//任务并行化数设为 10 个
numOfParallel = 2;
builder = new TopologyBuilder();
String desSpout = "my_spout";

```

```
String desBolt = "my_bolt";
//设置 spout 数据源
builder.setSpout(desSpout, new TestSpout(), numOfParallel);
builder.setBolt(desBolt, new TestBolt(),
numOfParallel) .shuffleGrouping(desSpout);
config = new Config();
config.setNumWorkers(numOfParallel);
config.setMaxSpoutPending(65536);
config.put(Config.STORM_ZOOKEEPER_CONNECTION_TIMEOUT, 40000);
config.put(Config.STORM_ZOOKEEPER_SESSION_TIMEOUT, 40000);
component2Node = new HashMap<>();
component2Node.put(desSpout, "special-supervisor1");
component2Node.put(desBolt, "special-supervisor2");
//此标识代表 topology 需要被调度
config.put("assigned_flag", "1");
//具体的组件节点对信息
config.put("design_map", component2Node);
StormSubmitter.submitTopology("test", config, builder.createTopology());
```

使用方法

打包此项目, 将 jar 包拷贝到 nimbus 节点的 STORM_HOME/lib 目录下。

在 nimbus 节点的 storm.yaml 配置中, 进行如下的配置:

```
storm.scheduler: "storm.DirectScheduler"
```

在 supervisor 节点中进行名称的配置, 配置项如下:

```
supervisor.scheduler.meta:
name: "your-supervisor-name"
```

总结:

DirectScheduler 把划分单位缩小到组件级别, 1 个 Spout 和 1 个 Bolt 可以指定到某个节点上运行, 如果没有指定, 还是按照系统自带的调度器进行调度。这个配置在 Topology 提交的 conf 配置中可进行设置。



活动 3.1: 优化活动 2.2 中的 WordCount Topology, 增加并行度, 通过 Storm UI 进行分析



活动 3.2：改变正在运行中的 **topology** 的并行度

NIIT | training.com-china

练习问题

1. 关于 storm 并行描述错误的是 ()
 - a. storm 的并行指的是由非常多的 supervisor 组成
 - b. 默认 1 个 work, 1 个 executor, 1 个 task
 - c. work 的数量是有配置文件中指定的
 - d. 以上都不对
2. 关于 work、executor、task 描述不正确的是 ()
 - a. 1 个或多个 executor 可能运行于 1 个单独的 worker 进程
 - b. 每 1 个 executor 从属于 1 个被 worker process 生成的线程中
 - c. 每 1 个 executor 运行着相同的组件(spout 或 bolt)的 1 个或多个 task
 - d. 以上都不对
3. 那种情况不是保障 storm 的消息可靠性 ()
 - a. 设置 Ackers 的数量为非 0:
`conf.setNumAckers(1);`
 - b. 更换一个可容错的高效的发射源
 - c. 在 spout 发射消息时, 绑定 messageId: `collector.emit(new Values(l), messageId);`
 - d. 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));`
4. 如何将 worker 数量设置为 2?
 - a. config.setNumberWorkers(2);
 - b. config.setNumWorker(2);
 - c. config.setBoltWorkers(2);
 - d. config.setNumWorkers(2);
5. 如何为 splitbolt 设置 4 个 task?
 - a. builder.setBolt(SPLIT_BOLT_ID, splitBolt, 4);
 - b. builder.setNumTasks(SPLIT_BOLT_ID, splitBolt, 4);
 - c. config.setBolt(SPLIT_BOLT_ID, splitBolt, 2);
 - d. builder.setBolt(SPLIT_BOLT_ID, splitBolt, 2).setNumTasks(4);

小结

在本章中，您学习到了：

- Thread(线程)可以在 worker 中运行，这些 thread 被称为执行器。在执行程序中，将运行 task。
- 总结一下， supervisor(节点)>worker(进程)>executor(线程)>task(实例)
- Storm 可以在不需要重启集群或拓扑，来增加或减少 worker 进程和 executor 的数量。这种方式叫 rebalancing。

```
$ storm rebalance mytopology -n 5 -e blue-spout=3 -e yellow-bolt=10
```

消息的可靠处理

- Storm 可靠性需要调用如下 2 个方法：
 - 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);`
- Topology 中 acker 任务的并行度可以通过配置参数 Config.TOPOLOGY_ACKERS 来设置。
- Task 级别失败：
 - bolt 任务失败
 - acker 任务失败
 - Spout 任务失败
- Scheduler 是 storm 的调度器，它负责为 Topology 分配当前的集群可用资源，目前 storm 提供了 3 种调度器：
 - EvenScheduler
 - DefaultScheduler
 - IsolationScheduler
- Storm 流式处理，需要在不同的业务场景中来配置 spout 和 bolt 的并行度，以达到提升系统处理速度的要求，但是需要了解并行度的概念。

Storm 高级应用

Storm 里面引入 DRPC 主要是利用 Storm 的实时计算能力来并行化 CPU 密集型 (CPU intensive) 的计算任务。DRPC 的 storm topology 以函数的参数流作为输入，而把这些函数调用的返回值作为 topology 的输出流。DRPC 其实不能算是 Storm 本身的一个特性，它是通过组合 Storm 的原语 stream、spout、bolt、topology 而成的一种模式 (pattern)。本来应该把 DRPC 单独打成一个包的，但是 DRPC 实在是太有用了，所以我们把它和 storm 捆绑在一起。

Trident 是 Storm 上实时计算的高级抽象。在数据需要只被处理一次的场景下是非常有用的。Trident 并不能对所有情况都适用，尤其是在需要高性能时，因为 Trident 添加了状态管理，使得程序的复杂性增加。

目标

在本章中，您将学习：

- DRPC 概述
- Trident 概述

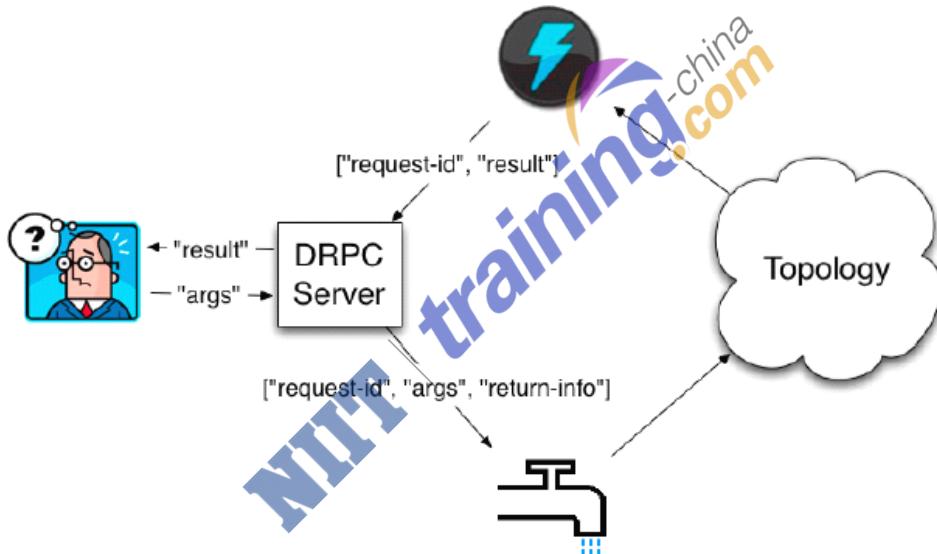


DRPC 概述

Distributed RPC 是由一个“DRPC 服务器”协调(storm 自带了一个实现)。DRPC 服务器协调：

- 接收一个 RPC 请求
- 发送请求到 storm topology
- 从 storm topology 接收结果
- 把结果发回给等待的客户端。
- 从客户端的角度来看一个 DRPC 调用跟一个普通的 RPC 调用没有任何区别

DRPC 工作流程



DRPC 工作流程

- 客户端给 DRPC 服务器发送要执行的函数 (function) 的名字，以及这个函数的参数。
- 实现了这个函数的 topology 使用 DRPCSpout 从 DRPC 服务器接收函数调用流，每个函数调用被 DRPC 服务器标记了一个唯一的 id。
- 这个 topology 然后计算结果，在 topology 的最后，一个叫做 ReturnResults 的 bolt 会连接到 DRPC 服务器，并且把这个调用的结果发送给 DRPC 服务器(通过那个唯一的 id 标识)。
- DRPC 服务器用那个唯一 id 来跟等待的客户端匹配上，唤醒这个客户端并且把结果发送给它。

Linear DRPC Topology Builder

Storm 自带了一个称作 LinearDRPCTopologyBuilder 的 topologybuilder，它把实现 DRPC 的几乎所有步骤都自动化了。这些步骤包括：

- 设置 spout
- 把结果返回给 DRPC 服务器
- 给 bolt 提供有限聚合几组 tuples 的能力

DRPC 简单例子

让我们看一个简单的例子。这是一个 DRPC 拓扑的实现，在输入参数的尾部追加“！”并返回：

```
public static class ExclaimBolt implements IBasicBolt {  
    public void prepare(Map conf, TopologyContext context){  
    }  
    public void execute(Tuple tuple, BasicOutputCollector collector) {  
        String input = tuple.getString(1);  
        collector.emit(new Values(tuple.getValue(0), input + "!"));  
    }  
    public void cleanup() {  
    }  
    public void declareOutputFields(OutputFieldsDeclarer declarer) {  
        declarer.declare(new Fields("id", "result"));  
    }  
}  
public static void main(String[] args) throws Exception {  
    LinearDRPCTopologyBuilder builder = new  
    LinearDRPCTopologyBuilder("exclamation");  
    builder.addBolt(new ExclaimBolt(), 3);  
    // ...  
}
```

一个 DRPC 服务器可以协调许多 Function，Function 名称用于区别不同的 Function。

首先声明的 bolt 将接收一个输入的 2-tuples，第一个 field 是请求 ID，第二个 field 是请求参数。LinearDRPCTopologyBuilder 认为最后的 bolt 会发射一个输出流，该输出流包含[id, result]格式的 2-tuples。

最后，所有拓扑中间过程产生的元组（tuple）都包含请求 id 作为其第一个 field。在这个例子中，ExclaimBolt 只是简单地在元组的第二个 field 尾部追加“！”字符。LinearDRPCTopologyBuilder 处理其余的协调工作，包括连接 DRPC 服务器，发送最终结果。

本地模式 DRPC

DRPC 可以运行在本地模式。这是如何在本地模式运行上述例子：

```
LocalDRPC drpc = new LocalDRPC();
```

```
LocalCluster cluster = new LocalCluster();
cluster.submitTopology("drpc-demo", conf, builder.createLocalTopology(drpc));
System.out.println("Results for 'hello': " + drpc.execute("exclamation", "hello"));
cluster.shutdown();
drpc.shutdown();
```

首先你创建一个 LocalDRPC 对象。这个对象在进程内模拟一个 DRPC 服务器，就像在进程内模拟一个 storm 集群一样。然后你创建本地集群，在本地模式运行这个拓扑。创建本地拓扑和远程拓扑，LinearDRPCTopologyBuilder 有不同的方法。在本地模式，LocalDRPC 未绑定任何端口，拓扑也需要知道与哪个对象通讯，这是为什么 createLocac1Topology 方法需要接受 LocalDRPC 对象作为输入参数的原因。

载入拓扑后，你可以用 LocalDRPC 的 execute 方法执行 DRPC 调用

远程模式 DRPC

在实际的集群使用 DRPC 也很简单。有三个步骤：

1. 启动 DRPC 服务器

使用 storm 脚本启动 DRPC 服务器，和启动 nimbus 和 ui 一样；

```
bin/storm drpc
```

2. 配置 DRPC 服务器位置

接下来，配置你的 storm 集群，让集群知道 DRPC 服务器的位置，这样 DRPCSpout 就知道从 里读取功能调用。可以通过修改 storm.yaml 配置文件或拓扑配置完成配置 DRPC 服务器位置。修改 storm.yaml 配置文件如下所示：

```
drpc.servers:
  - "drpc1.foo.com"
  - "drpc2.foo.com"
```

提交 DRPC 拓扑到 storm 集群

3. 最后，使用 StormSubmitter 启动 DRPC 拓扑，就像启动其它拓扑一样。在远程模式运行上述例子，代码如下所示：

```
StormSubmitter.submitTopology("exclamation-drpc",
    conf,
    builder.createRemoteTopology());
```

createRemoteTopology 方法用于在 storm 集群创建拓扑。

DRPC 复杂例子

针对 twitter 网站上的一个 URL 的接触用户进行统计。

一个 URL 的接触用户数是在 twitter 网站上接触一个 URL 的用户数，你需要以下 4 步：

1. 获取 tweeted the URL 的全部用户

2. 获取这些用户的全部追随者
3. 使追随者集合中的用户唯一
4. 统计唯一的用户数

一个单独的 reach 计算在计算期间涉及到数千数据库访问和数千万追随者记录。它是一个真正的耗时计算。正如你将要看到的，在 storm 上实现这个功能非常简单。在一台机器上，reach 计算花费数分钟，在 storm 集群，最难计算 reach 的 URL 也只需数秒。

Storm-starter 项目里定义了一个 reach 样例，reach 拓扑定义如下所示：

```
LinearDRPCTopologyBuilder builder = new LinearDRPCTopologyBuilder("reach");  
builder.addBolt(new GetTweeters(), 3);  
builder.addBolt(new GetFollowers(), 12) .shuffleGrouping();  
builder.addBolt(new PartialUniquer(), 6) .fieldsGrouping(new Fields("id",  
"follower"));  
builder.addBolt(new CountAggregator(), 2) .fieldsGrouping(new Fields("id"));
```

这个拓扑以 4 个步骤的形式执行：

1. GetTweeters 获取 tweeted the URL 的用户。它转换一个[id, url]形式的输入流到[id, tweeter]形式的输出流。每个 url 元组将映射到多个 tweeter 元组。
2. GetFollowers 获取这些 tweeter 的追随者。它转换一个[id, tweeter]形式的输入流到[id, follower]形式的输出流。跨所有任务，当某人追随多个 tweeter，这些 tweeter 又 tweeted 相同的 URL 时，这可能会得到重复的追随者。
3. PartialUniquer 按追随者 ID 对追随者数据流进行分组。同一的追随者去到同一的任务，因此每个 PartialUniquer 任务都接收到独立的相互独立的追随者集合。PartialUniquer 一旦收到请求 ID 用于它的所有追随者元组，它就发射追随者子集的唯一总数。
4. 最后，CountAggregator 从每个 PartialUniquer 任务接收计数并对它们求和。

代码如下：

```
import backtype.storm.Config;  
import backtype.storm.LocalCluster;  
import backtype.storm.LocalDRPC;  
import backtype.storm.StormSubmitter;  
import backtype.storm.coordination.BatchOutputCollector;  
import backtype.storm.drpc.LinearDRPCTopologyBuilder;  
import backtype.storm.task.TopologyContext;  
import backtype.storm.topology.BasicOutputCollector;  
import backtype.storm.topology.OutputFieldsDeclarer;  
import backtype.storm.topology.base.BaseBasicBolt;  
import backtype.storm.topology.base.BaseBatchBolt;  
import backtype.storm.tuple.Fields;  
import backtype.storm.tuple.Tuple;  
import backtype.storm.tuple.Values;  
import java.util.*;  
public class ReachTopology {  
    public static Map<String, List<String>> TWEETERS_DB = new HashMap<String,  
List<String>>() {{  
        put("foo.com/blog/1", Arrays.asList("sally", "bob", "tim", "george",  
"nathan"));  
    }}  
}
```

```

        put("engineering.twitter.com/blog/5", Arrays.asList("adam", "david", "sally",
    "nathan"));
        put("tech.backtype.com/blog/123", Arrays.asList("tim", "mike", "john"));
    }};
    public static Map<String, List<String>> FOLLOWERS_DB = new HashMap<String,
List<String>>() {{
        put("sally", Arrays.asList("bob", "tim", "alice", "adam", "jim", "chris",
    "jai"));
        put("bob", Arrays.asList("sally", "nathan", "jim", "mary", "david",
    "vivian"));
        put("tim", Arrays.asList("alex"));
        put("nathan", Arrays.asList("sally", "bob", "adam", "harry", "chris",
    "vivian", "emily", "jordan"));
        put("adam", Arrays.asList("david", "carissa"));
        put("mike", Arrays.asList("john", "bob"));
        put("john", Arrays.asList("alice", "nathan", "jim", "mike", "bob"));
    }};
public static class GetTweeters extends BaseBasicBolt {
    public void execute(Tuple tuple, BasicOutputCollector collector) {
        Object id = tuple.getValue(0);
        String url = tuple.getString(1);
        List<String> tweeters = TWEETERS_DB.get(url);
        if (tweeters != null) {
            for (String tweeter : tweeters) {
                collector.emit(new Values(id, tweeter));
            }
        }
    }
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("id", "tweeter"));
    }
}
public static class GetFollowers extends BaseBasicBolt {
    public void execute(Tuple tuple, BasicOutputCollector collector) {
        Object id = tuple.getValue(0);
        String tweeter = tuple.getString(1);
        List<String> followers = FOLLOWERS_DB.get(tweeter);
        if (followers != null) {
            for (String follower : followers) {
                collector.emit(new Values(id, follower));
            }
        }
    }
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("id", "follower"));
    }
}
public static class PartialUniquer extends BaseBatchBolt {
    BatchOutputCollector _collector;
    Object _id;
    Set<String> _followers = new HashSet<String>();
    public void prepare(Map conf, TopologyContext context, BatchOutputCollector
collector, Object id) {
        _collector = collector;
        _id = id;
    }
}

```

```

    }
    public void execute(Tuple tuple) {
        _followers.add(tuple.getString(1));
    }
    public void finishBatch() {
        _collector.emit(new Values(_id, _followers.size()));
    }
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("id", "partial-count"));
    }
}
public static class CountAggregator extends BaseBatchBolt {
    BatchOutputCollector _collector;
    Object _id;
    int _count = 0;
    public void prepare(Map conf, TopologyContext context, BatchOutputCollector
    collector, Object id) {
        _collector = collector;
        _id = id;
    }
    public void execute(Tuple tuple) {
        _count += tuple.getInteger(1);
    }
    public void finishBatch() {
        _collector.emit(new Values(_id, _count));
    }
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("id", "reach"));
    }
}
public static LinearDRPCTopologyBuilder construct() {
    LinearDRPCTopologyBuilder builder = new LinearDRPCTopologyBuilder("reach");
    builder.addBolt(new GetTweeters(), 4);
    builder.addBolt(new GetFollowers(), 12).shuffleGrouping();
    builder.addBolt(new PartialUniquer(), 6).fieldsGrouping(new Fields("id",
    "follower"));
    builder.addBolt(new CountAggregator(), 3).fieldsGrouping(new Fields("id"));
    return builder;
}
public static void main(String[] args) throws Exception {
    LinearDRPCTopologyBuilder builder = construct();
    Config conf = new Config();
    if (args == null || args.length == 0) {
        conf.setMaxTaskParallelism(3);
        LocalRPC drpc = new LocalRPC();
        LocalCluster cluster = new LocalCluster();
        cluster.submitTopology("reach-drpc", conf,
builder.createLocalTopology(drpc));
        String[] urlsToTry = new String[] { "foo.com/blog/1",
"engineering.twitter.com/blog/5", "notaurl.com" };
        for (String url : urlsToTry) {
            System.out.println("Reach of " + url + ": " + drpc.execute("reach", url));
        }
        cluster.shutdown();
        drpc.shutdown();
    }
}

```

```
        else {
            conf.setNumWorkers(6);
            StormSubmitter.submitTopology(args[0], conf,
builder.createRemoteTopology());
        }
    }
}
```

当 PartialUniquer 在 execute 方法中接收一个 follower 元组时，它用一个内部 HashMap 添加它到与请求 ID 对应的集合。

PartialUniquer 也实现了 FinishedCallback 接口，它告诉 LinearDRPCTopologyBuilder，对于任意给定的请求 ID，当它已收到所有指向它的元组时，请通知它。这个回调是 finishedId 方法。在这个回调中，PartialUniquer 发射单一的元组，元组包含它的追随者子集的唯一总数。

在底层，CoordinatedBolt 用于检测一个 bolt 何时收到该请求 ID 的所有元组。CoordinatedBolt 使用 direct stream 管理协调。

其它的拓扑应该是不言自明。如你所见，reach 计算的每一单步都是并行执行的，而且定义一个 DRPC 拓扑也非常简单。



活动 4.1：使用 DRPC Topology 判断输入参数是否为质数

Trident 概述

Trident 是对 Storm 的更高一层的抽象,除了提供一套简单易用的流数据处理 API 之外, 它以 batch(一组 tuples)为单位进行处理, 这样一来, 可以使得一些处理更简单和高效。

我们知道把 Bolt 的运行状态仅仅保存在内存中是不可靠的, 如果一个 node 挂掉, 那么这个 node 上的任务就会被重新分配, 但是之前的状态是无法恢复的。因此, 比较聪明的方式就是把 storm 的计算状态信息持久化到 database 中, 基于这一点, trident 就变得尤为重要。因为在处理大数据时, 我们在与 database 打交道时通常会采用批处理的方式来避免给它带来压力, 而 trident 恰恰是以 batch groups 的形式处理数据, 并提供了一些聚合功能的 API。

Trident 有五类操作包括如下:

1. Partition-local operations: 对每个 partition 的局部操作, 不产生网络传输
2. Repartitioning operations: 对数据流的重新划分(仅仅是划分, 但不改变内容), 产生网络传输
3. Aggregation operations: 聚合操作
4. Operations on grouped streams: 作用在分组流上的操作
5. Merge、Join 操作

以下对 5 类操作的详细解释

Partition 操作

对每个分区(partition)的局部操作有: function、filter、project 等。

■ Function

一个 function 收到一个输入 tuple 后可以输出 0 或多个 tuple, 输出 tuple 的字段被追加到接收到的输入 tuple 后面。如果对某个 tuple 执行 function 后没有输出 tuple, 则该 tuple 被过滤(filter), 否则, 就会为每个输出 tuple 复制一份输入 tuple 的副本。假设有如下的 function:

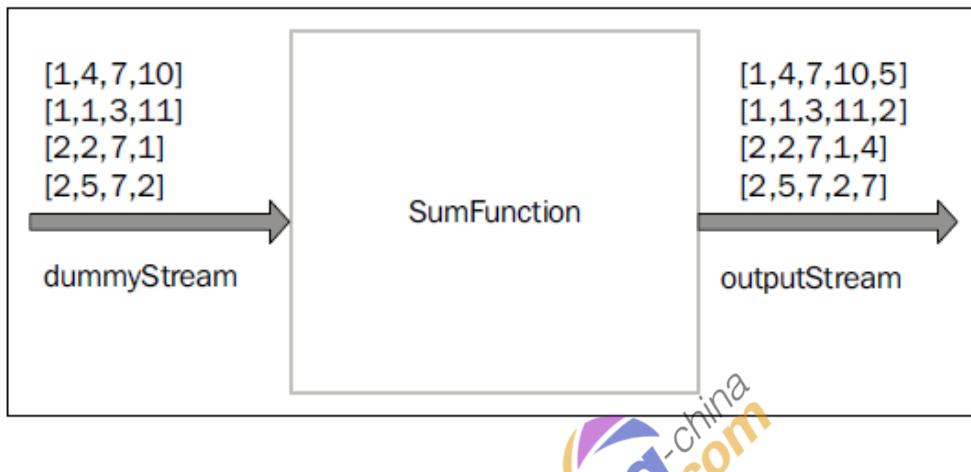
```
class SumFunction extends BaseFunction {  
    private static final long serialVersionUID = 5L;  
    public void execute(TridentTuple tuple, TridentCollector collector) {  
        int number1 = tuple.getInteger(0);  
        int number2 = tuple.getInteger(1);  
        int sum = number1 + number2;  
        // emit the sum of first two fields  
        collector.emit(new Values(sum));  
    }  
}
```

假设有个叫“mystream”的流(stream), 该流中有如下 tuple (tuple 的字段为["a", "b", "c"]), 运行下面的代码:

```
DiagnosisEventSpout spout = new DiagnosisEventSpout();  
Stream mystream = topology.newStream("event", spout);
```

```
mystream.each(new Fields("a","b"), new SumFunction (), new Fields("sum"))
```

则输出 tuple 中的字段为["a", "b", "c", "d"]，如下所示



■ Filter

根据一个字段过滤一个行是否保留，需要实现 BaseFilter 接口，并重写方法

```
public boolean isKeep(TridentTuple tuple) {  
    // TODO Auto-generated method stub  
    return false;  
}
```

如果返回 true 表示保留此行，返回 false 表示过滤此行

```
static class CheckEvenSumFilter extends BaseFilter {  
    private static final long serialVersionUID = 7L;  
    public boolean isKeep(TridentTuple tuple) {  
        int number1 = tuple.getInteger(0);  
        int number2 = tuple.getInteger(1);  
        int sum = number1 + number2;  
        if (sum % 2 == 0) {  
            return true;  
        }  
        return false;  
    }  
}
```

假设有个叫 “mystream” 的流(stream)，该流中有如下 tuple (tuple 的字段为["a", "b", "c", "d"])，运行下面的代码：

```
mystream.each(new Fields("a","b"), new CheckEvenSumFilter ())
```



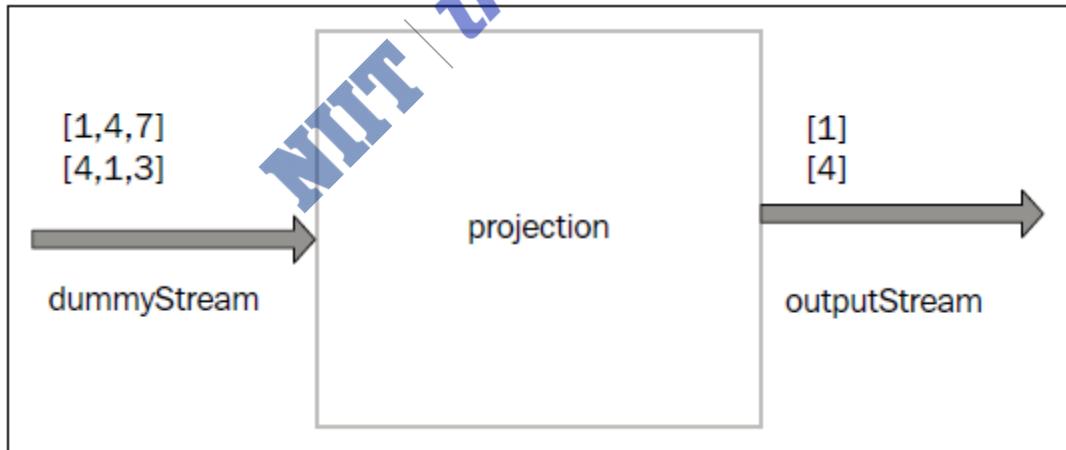
Filter

■ Projection

经 Stream 中的 project 方法处理后的 tuple 仅保持指定字段（相当于过滤字段）。例如，
mystream 中的字段为 ["a", "b", "c", "d"]，执行下面代码

```
mystream.project(new Fields("x"));
```

结果如下：

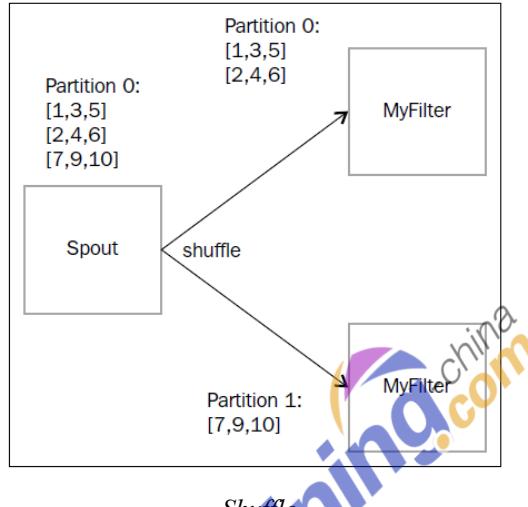


Project

Repartitioning 操作

Repartition 操作可以改变 tuple 在各个 task 之上的划分。Repartition 也可以改变 Partition 的数量。Repartition 需要网络传输。下面都是 repartition 操作：

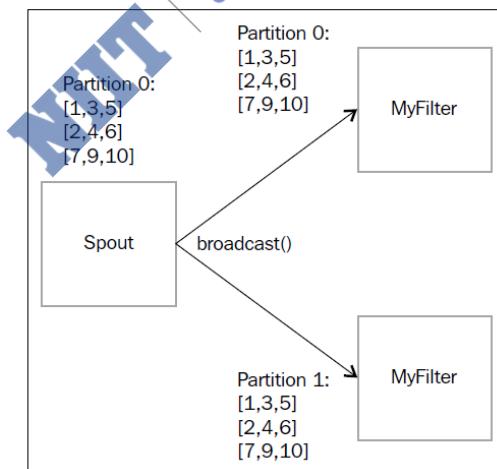
1. Shuffle: 随机将 tuple 均匀地分发到目标 partition 里。



Shuffle

```
mystream.shuffle().each(new Fields("a","b"), new  
myFilter()).parallelismHint(2)
```

2. Broadcast: 每个 tuple 被复制到所有的目标 partition 里



Broadcast

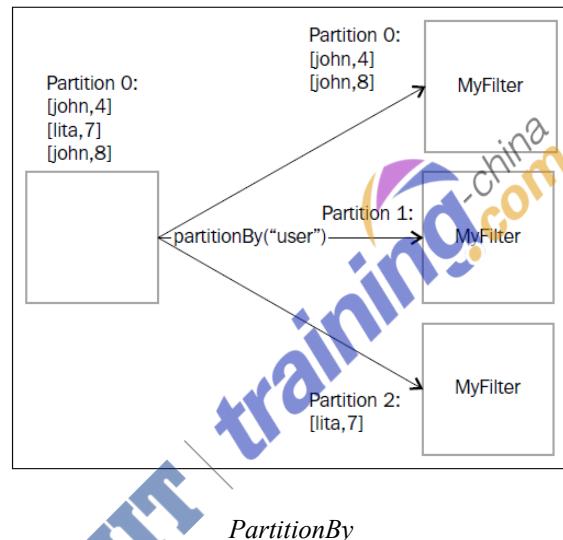
```
mystream.broadcast().each(new Fields("a","b"),new  
myFilter()).parallelismHint(2)
```

3. PartitionBy: 对每个 tuple 选择 partition 的方法是: (该 tuple 指定字段的 hash 值) mod (目标 partition 的个数), 该方法确保指定字段相同的 tuple 能够被发送到同一个 partition。(但同一个 partition 里可能有字段不同的 tuple)

```
mystream.partitionBy(new Fields("username")).each(new  
Fields("username", "text"), new myFilter()).parallelismHint(2)
```

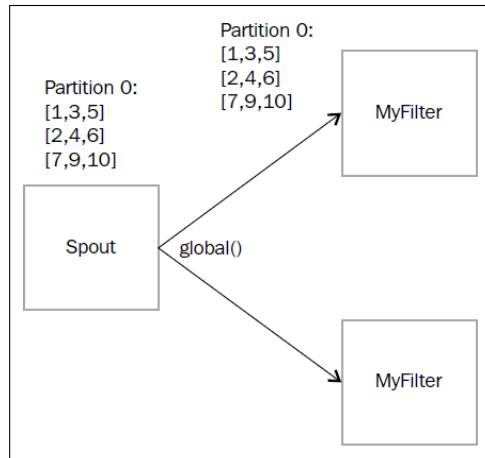
目标 partition 的计算方式

target partition = hash (fields) % (number of target partition)



4. global: 所有的 tuple 都被发送到同一个 partition。

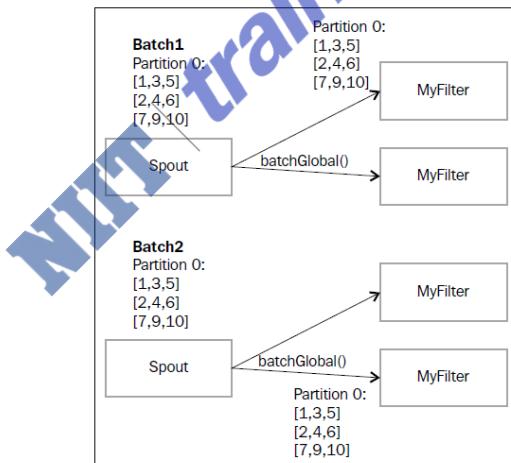
```
mystream.global().each(new Fields("a", "b"), new  
myFilter()).parallelismHint(2)
```



Global

5. batchGlobal: 确保同一个 batch 中的 tuple 被发送到相同的 partition 中。

```
mystream.batchGlobal().each(new Fields("a", "b"), new  
myFilter()).parallelismHint(2);
```



batchGlobal

代码示例:

1. each() 方法

操作 batch 中的每一个 tuple 内容，一般与 Filter 或者 Function 函数配合使用。

2. parallelismHint()

指定 Topology 的并行度，即用多少线程执行这个任务。我们可以稍微改一下我们的 Filter，通过打印当前任务的 partitionIndex 来区分当前是哪个线程。

3. Filter

```
public static class PerActorTweetsFilter extends BaseFilter {  
    private int partitionIndex;  
    private String actor;  
    public PerActorTweetsFilter(String actor) {  
        this.actor = actor;  
    }  
    @Override  
    public void prepare(Map conf, TridentOperationContext context) {  
        this.partitionIndex = context.getPartitionIndex();  
    }  
    @Override  
    public boolean isKeep(TridentTuple tuple) {  
        boolean filter = tuple.getString(0).equals(actor);  
        if(filter) {  
            System.err.println("I am partition [" + partitionIndex + "] and I  
have kept a tweet by: " + actor);  
        }  
        return filter;  
    }  
}
```

4. Topology:

```
topology.newStream("spout", spout)  
.each(new Fields("actor", "text"), new PerActorTweetsFilter("dave"))  
.parallelismHint(5)  
.each(new Fields("actor", "text"), new Utils.PrintFilter());
```

如果我们指定执行 Filter 任务的线程数量为 5，那么最终的执行结果会如何呢？看一下我们的测试结果：

```
I am partition [4] and I have kept a tweet by: dave  
I am partition [3] and I have kept a tweet by: dave  
I am partition [0] and I have kept a tweet by: dave  
I am partition [2] and I have kept a tweet by: dave  
I am partition [1] and I have kept a tweet by: dave
```

我们可以很清楚的发现，一共有 5 个线程在执行 Filter。

如果我们想要 2 个 Spout 和 5 个 Filter 怎么办呢？如下面代码所示，实现很简单。

```
topology.newStream("spout", spout)  
.parallelismHint(2)  
.shuffle()  
.each(new Fields("actor", "text"), new PerActorTweetsFilter("dave"))  
.parallelismHint(5)
```

```
.each(new Fields("actor", "text"), new Utils.PrintFilter());  
5. partitionBy()和重定向操作(repartitioning operation)
```

我们注意到上面的例子中用到了 shuffle(), shuffle()是一个重定向操作。那什么是重定向操作呢？重定向定义了我们的 tuple 如何被 route 到下一处理层，当然不同的层之间可能会有不同的并行度，shuffle()的作用是把 tuple 随机的 route 到下一层的线程中，而 partitionBy()则根据我们的指定字段按照一致性哈希算法 route 到下一层的线程中，也就是说，如果我们用 partitionBy()的话，同一个字段名的 tuple 会被 route 到同一个线程中。

Aggregation 操作

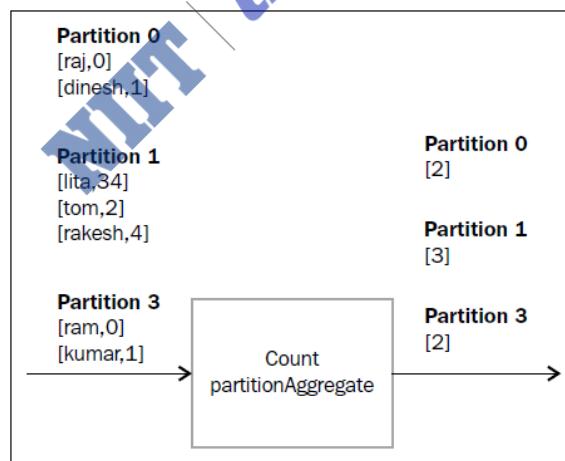
我们前面讲过，Trident 的一个很重要的特点就是它是以 batch 的形式处理 tuple 的。我们可以很容易想到的针对一个 batch 的最基本操作应该就是聚合。Trident 提供了聚合 API 来处理 batches，Trident 中有 3 种聚合的方式：

- partition aggregate
- aggregate
- persistence aggregate

partitionAggregate

partitionAggregate 对每个 partition 执行一个 function 操作（实际上是聚合操作），但它又不同于上面的 function 操作，partitionAggregate 的输出 tuple 将会取代收到的输入 tuple，如下面的例子：

```
mystream.partitionAggregate(new Fields("x"), new Count(), new Fields("count"));
```



partitionAggregate

Aggregate

有 3 种 aggregate 类型：

- ReducerAggregator

```
public interface ReducerAggregator extends Serializable {  
    T init();  
    T reduce(T curr, TridentTuple tuple);  
}
```

ReducerAggregator 使用 init()方法产生一个初始值，对于每个输入 tuple，依次迭代这个初始值，最终产生一个单值输出 tuple。例如下列代码：

```
public class Count implements ReducerAggregator {  
    public Long init() {  
        return 0L;  
    }  
  
    public Long reduce(Long curr, TridentTuple tuple) {  
        return curr + 1;  
    }  
}
```

- Aggregator

通用聚合框架 Aggregator，需要实现 3 个方法，如下描述

```
public interface Aggregator extends Operation {  
    T init(Object batchId, TridentCollector collector);  
    void aggregate(T state, TridentTuple tuple, TridentCollector collector);  
    void complete(T state, TridentCollector collector);  
}
```

Aggregator 可以输出任意数量的 tuple，且这些 tuple 的字段也可以有多个。执行过程中的任何时候都可以输出 tuple（三个方法的参数中都有 collector）。Aggregator 的执行方式如下：

1. 处理每个 batch 之前调用一次 init()方法，该方法的返回值是一个对象，代表 aggregation 的状态，并且会传递给下面的 aggregate()和 complete()方法。
2. 每个收到一个该 batch 中的输入 tuple 就会调用一次 aggregate，该方法中可以更新状态（第一点中 init()方法的返回值）。
3. 当该 batch partition 中的所有 tuple 都被 aggregate()方法处理完之后调用 complete 方法。

```
public class CountAgg extends BaseAggregator {  
    static class CountState {  
        long count = 0;  
    }  
    public CountState init(Object batchId, TridentCollector collector) {  
        return new CountState();  
    }  
    public void aggregate(CountState state, TridentTuple tuple,  
    TridentCollector collector) {  
        state.count+=1;  
    }  
    public void complete(CountState state, TridentCollector collector) {  
        collector.emit(new Values(state.count));  
    }  
}
```

总结：通用聚合方法 Aggregator

- a. 使用一个内部类来保存当前的状态
- b. 跟在 streamGroup 后面
- c. CombinerAggregator

```
public interface CombinerAggregator extends Serializable {  
    T init(TridentTuple tuple);  
    T combine(T val1, T val2);  
    T zero();  
}
```

一个 CombinerAggregator 仅输出一个 tuple（该 tuple 也只有一个字段）。每收到一个输入 tuple，CombinerAggregator 就会执行 init()方法（该方法返回一个初始值），并且用 combine()方法汇总这些值，直到剩下一个值为止（聚合值）。如果 partition 中没有 tuple，CombinerAggregator 会发送 zero()的返回值。

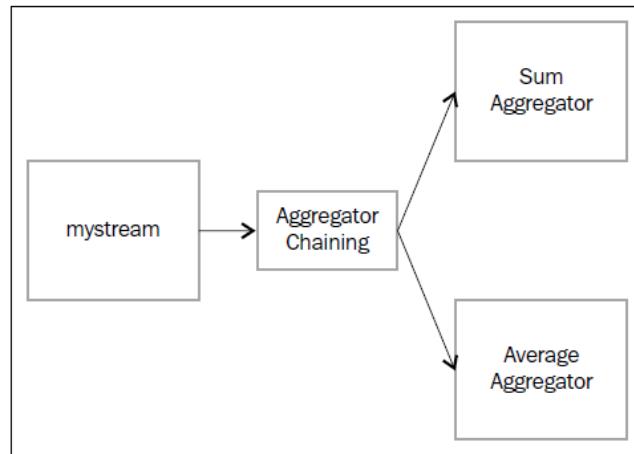
```
public class Count implements CombinerAggregator {  
    public Long init(TridentTuple tuple) {  
        return 1L;  
    }  
    public Long combine(Long val1, Long val2) {  
        return val1 + val2;  
    }  
    public Long zero() {  
        return 0L;  
    }  
}
```

■ Aggregator Chaining

有时需要同时执行多个聚合操作，这个可以使用链式操作完成

```
mystream.chainedAgg()  
.partitionAggregate(new Average(), new Fields("Average"))  
.partitionAggregate(new Fields("b"), new Sum(), new Fields("sum"))  
.chainEnd()
```

这段代码将会对每个 partition 执行 Average 和 Sum 聚合器，并输出一个 tuple（字段为 ["Average", "sum"]）。



Aggregator Chaining

persistenceAggregate

partitionAggregate 通常用于 global aggregation 时的本地化聚合，类似于 Hadoop 中的 map 阶段。partitionAggregate 是在每一个 partition 内独立调用自己的聚合操作，互不干涉。最后还需要把局部聚合值 emit 出来，通过网络传输供后面的 aggregate 做全局聚合。通过这种策略，可以实现 global aggregation 的并发。partitionAggregate 的前面不能跟 groupBy 方法，因为 groupBy 方法返回的 GroupedStream 对象没有 partitionAggregate 方法。partitionAggregate 虽然也是聚合操作，但与上面的 Aggregate 完全不同，它不是一个重定向操作。partitionAggregate 对每个 partition 执行一个 function 操作（实际上是聚合操作），但它又不同于上面的 function 操作，partitionAggregate 的输出 tuple 将会取代收到的输入 tuple。

```
mystream.partitionAggregate(new Fields("b"), new Sum(), new Fields("sum"))
```

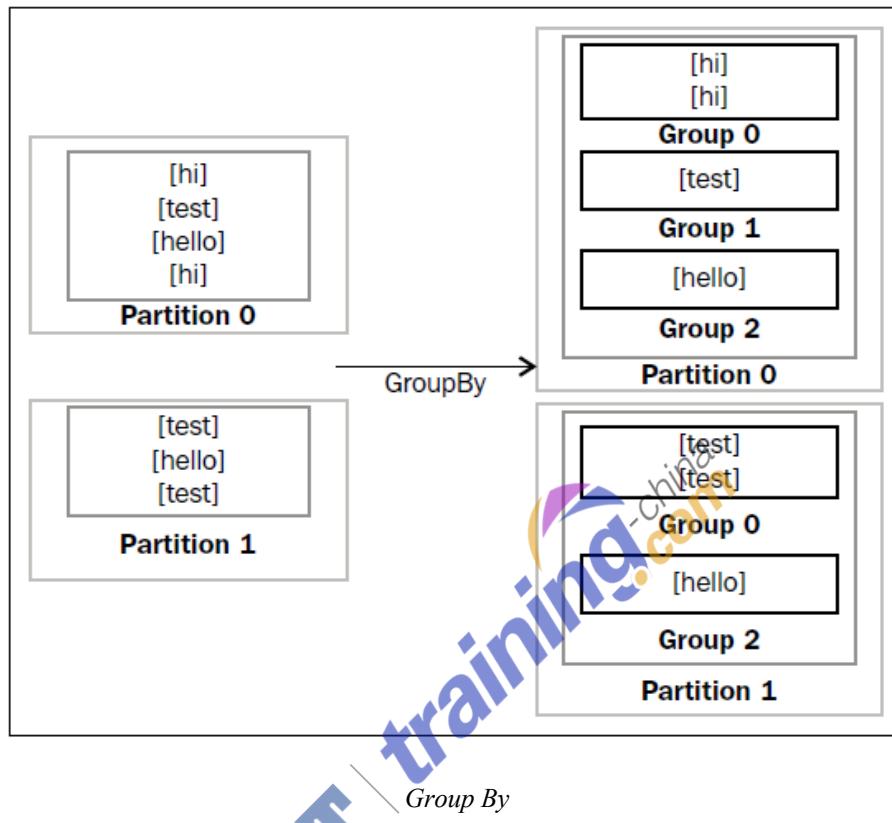
Group 操作

groupBy : groupBy 操作先对流中的指定字段做 partitionBy 操作，让指定字段相同的 tuple 能被发送到同一个 partition 里。然后在每个 partition 里根据指定字段值对该分区里的 tuple 进行分组。下面演示了 groupBy 操作的过程：

```
topology.newStream("spout", spout)
    .groupBy(new Fields("location"))
        .aggregate(new Fields("location"), new Count(), new Fields("count"))
            .each(new Fields("location", "count"), new Utils.PrintFilter());
```

上面这段代码计算出了每个 location 的数量，即使我们的 Count 函数没有指定并行度。这就是 groupBy()起的作用，它会根据指定的字段创建一个 GroupedStream，相同字段的 tuple 都会被重定向到一起，汇聚成一个 group。groupBy()之后是 aggregate，与之前的聚合整个 batch 不同，此时的

aggregate 会单独聚合每个 group。我们也可以这么认为，groupBy 会把 Stream 按照指定字段分成一个个 stream group，每个 group 就像一个 batch 一样被处理。



Merge 和 Join 操作

有两种方式可以合并 stream

- Merge : 可以将多个 stream 合并成一个 stream
`topology.merge(stream1, stream2, stream3);`
- Join : 使用 join 合并多个 stream。对一个标准的 join 连接来说，就像 SQL 里的 join，要求接受的输入数据是有限的，因此无法处理无限的数据流。Trident 里面的 join 只能小批量的处理从 spout 发出的数据。当来自不同 spout 的数据之间发生 join 时，这些 spout 之间将按照它们发射批次数据的顺序进行同步。所以，一个批处理过程将会包含每个 spout 中的所有 tuple.

这里有一个的例子，用来 join 一个包含字段 ["key1", "Product", "Price"] 的 stream 和另一个包含 ["key2", "Qty", "date"] 的 stream:

```
topology.join(stream1, new Fields("key1"), stream2, new Fields("key2"), new Fields("key", "a", "b", "c", "d"));
```

把"key1" 和"key2" 作为 join 的字段将 stream1 和 stream2 这两个流进行合并。之后 Trident 要求为合并后的流的所有字段设置名称，由于输入流之间有相同的字段，最终 join 之后发射的 tuple 将包含：

1. 首先包含 join 的字段列表。这个例子里面是"key"，对应的 stream1 里面的"key1" 和 stream2 里面的 "key2" 。
2. 其次包含非 join 的字段列表。顺序就是每个流被传入 join 方法的次序。在这个例子里， "a" 和"b" 分别对应 stream1 里的 "Product" 和 "Price"， "c" and "d"分别对应 stream2 里面的"Qty" 和 "Date" 。

 **Note**

Join 操作通过检查元组的字段来合并两个 stream。



活动 4.2：使用 Trident Topology 批量处理实时数据

练习问题

1. 关于 storm DRPC 描述错误的是（）
 - a. storm 接收一个 RPC 请求并发送请求到 storm topology
 - b. 从 storm topology 接收结果把结果发回给等待的客户端
 - c. 适合执行并行查询请求，串行的不适合使用 DRPC 来查询
 - d. 以上都不对
2. 关于 storm trident 描述不正确的是（）
 - a. Trident 是对 Storm 的更高一层的抽象,除了提供一套简单易用的流数据处理 API 之外，它以 batch(一组 tuples)为单位进行数据处理
 - b. trident 可以是以 batch group 的形式处理数据，并提供了一些聚合功能的 API
 - c. trident 中的消息可以是容错的，也可以是不容错的
 - d. 以上都不对
3. 那种情况不是 trident 的常用操作（）
 - a. Partition-local operations
 - b. Repartitioning operations
 - c. Dispatch operations
 - d. Aggregation operations
4. 下列哪种描述是错误的（）
 - a. Filter 需要实现 BaseFilter 接口，并实现 isKeep 方法
 - b. mystream.project(new Fields("x"))表示去掉字段为 x 的数据
 - c. function 需要实现 BaseFunction 接口，并实现 execute 方法
 - d. mystream.shuffle().each(new Fields("a","b"), new myFilter()).parallelismHint(2) 表示字段 ab 随机发到 myFilter 函数上
5. 下面关于 Trident 的说法中哪个是不正确的?
 - a. 是高级抽象
 - b. 保证数据只能处理一次
 - c. 高性能
 - d. 要实现 function 或 filter 需要分别继承 BaseFunction 类或 BaseFilter 类

小结

本章您学习了：

- DRPC 简介
 - DRPC 本地模式
 - DRPC 远程模式
- Trident 有五种操作，分别是：
 - Partition-local 操作：本地的分区操作不会发生网络数据传输
 - Repartitioning 操作：对数据流的 repartitioning 操作将产生网络数据传输(只分区，不改变内容)
 - Aggregation 操作
 - Group 操作
 - Merge 和 Join 操作
- Trident 使用 aggregation API 来处理批量数据，有三种方式：
 - partition aggregate
 - aggregate
 - persistence aggregate
- DRPC 以及 Trident 都是在 Storm 中非常重要的功能，DRPC 主要用来执行并发查询，将单线的查询转为并行查询，并自动合并查询结果用来提高查询时间。Trident 将单条处理的消息变成批量处理，增加了处理速度。
- 合并流的方式有两种：merge 和 join。

Trident 状态管理

Trident 是在 storm 基础上，一个以 real-time 计算为目标的高度抽象。Trident 提供 State 来持久化数据。

Trident 是读写有状态资源的一级抽象。状态既可以保存在内部 - 例如保存在内存中或者备份在 HDFS 文件系统中，也可以保存在外部 - 比如 Memcached 或者 Cassandra 中。无论是哪种情况使用的都是同样的 Trident API。

Trident 以容错的方式管理状态，因此状态更新在面对重试和失败时是幂等的。可以断定 Trident 拓扑的每个消息都能只会被精确处理一次。

目标

在本章中，您将学习：

- 可容错 spout
- State apis
- PersistentAggregate
- 实现 Map States
- 将 Trident 状态持久化到数据库



Trident 状态

Trident 在读写有状态的数据源方面是有着一流的抽象封装的。状态既可以保留在 topology 的内部，比如说内存和 HDFS，也可以放到外部存储当中，比如说 Memcached 或者 Cassandra。这些都是使用同一套 Trident API。

Trident 以一种容错的方式来管理状态以至于当你在更新状态的时候你不需要去考虑错误以及重试的情况。这种保证每个消息被处理有且只有一次的原理会让你更放心的使用 Trident 的 topology。

在进行状态更新时，会有不同的容错级别。在讨论这点之前，让我们先通过一个例子来说明一下如何达到有且只有一次处理的必要的技巧。假定你在做一个关于某 stream 的计数聚合器，你想要把运行中的计数存放到一个数据库中。如果你在数据库中存了一个值表示这个计数，每次你处理一个 tuple 之后，就将数据库存储的计数加一。

当错误发生，tuple 会被重播。这就带来了一个问题：当状态更新的时候，你完全不知道你是不是在之前已经成功处理过这个 tuple。也许你之前从来没处理过这个 tuple，这样的话你就应该把 count 加一。另外一种可能就是你之前是成功处理过这个 tuple 的，但是这个在其他的步骤处理这个 tuple 的时候失败了（如 ack 丢失），在这种情况下，我们就不应该将 count 加一。再或者，你接受到过这个 tuple，但是上次处理这个 tuple 的时候在更新数据库的时候失败了，这种情况你就应该去更新数据库。

如果只是简单的存计数到数据库的话，你是完全不知道这个 tuple 之前是否已经被处理过了的。所以你需要更多的信息来做正确的决定。Trident 提供了下面的语义来实现有且只有一次被处理的目标，并提供多种可容错 spout；

可容错 Spout

Trident 提供了下面的语义来实现有且只有一次被处理的目标：

1. Tuples 是被分成小的集合（一组 tuple 被称为一个 batch）被批量处理的。
2. 每一批 tuples 被给定一个唯一 ID 作为事务 ID (txid). 当这一批 tuple 被重播时, txid 不变。
3. 批与批之间的状态更新时严格顺序的。比如说第三批 tuple 的状态的更新必须要等到第二批 tuple 的状态更新成功之后才可以进行。

有了这些定义，你的状态实现可以检测到当前这批 tuple 是否以前处理过，并根据不同的情况进行不同的处理，这个处理取决于你的输入 spout。有三种不同类型的可以容错的 spout: “non-transactional”，“transactional” 和 “opaque transactional”。对应的，也有 3 种容错的状态：“non-transactional”，“transactional” 和 “opaque transactional”。让我们一起来看看每一种 spout 类型能够支持什么样的容错类型。

■ Non-transactional

Non-transactional Spout(非事务 Spout)不确保每个 batch 中的 tuple 的规则 (是否重叠)。所以如果 tuple 被处理失败不重发则该 tuple 最多被处理一次，如果 tuple 在不同的 batch 中被多次成功处理的时候他也可能会是至少处理一次的。无论怎样，这种 spout 是不可能实现有且只有一次被成功处理的语义的。

■ Transactional Spouts

Trident 是以小批量 (batch) 的形式在处理 tuple，并且每一批都会分配一个唯一的 transaction id。不同 spout 的特性不同，一个 transactional spout 会有如下这些特性：

1. 有着同样 txid 的 batch 一定是一样的。当重播一个 txid 对应的 batch 时，一定会重播和之前对应 txid 的 batch 中同样的 tuples。
2. 各个 batch 之间是没有交集的。每个 tuple 只能属于一个 batch
3. 每一个 tuple 都属于一个 batch，无一例外

这是一类非常容易理解的 spout，tuple 流被划分为固定的 batch 并且永不改变。（trident-kafka 有一个 transactional spout 的实现）

你也许会问：为什么我们不总是使用 transactional spout？这很容易理解。一个原因是并不是所有的地方都需要容错的。举例来说，TransactionalTridentKafkaSpout 工作的方式是一个 batch 包含的 tuple 来自某个 kafka topic 中的所有 partition。一旦这个 batch 被发出，在任何时候如果这个 batch 被重新发出时，它必须包含原来所有的 tuple 以满足 transactional spout 的语义。现在我们假定一个 batch 被 TransactionalTridentKafkaSpout 所发出，这个 batch 没有被成功处理，并且同时 kafka 的一个节点也 down 掉了。你就无法像之前一样重播一个完全一样的 batch（因为 kafka 的节点 down 掉，该 topic 的一部分 partition 可能会无法使用），整个处理会被中断。

这也就是”opaque transactional” spouts (不透明事务 spout) 存在的原因 – 他们对于丢失源节点这种情况是容错的，仍然能够帮你达到有且只有一次处理的语义。后面会对这种 spout 有所介绍。

在讨论“opaque transactional” spout 之前，我们先来看看怎样为 transactional spout 设计一个具有 exactly-once 语义的 State 实现。这个 State 的类型是“transactional state”并且它利用了任何一个 txid 总是对应同样的 tuple 序列这个语义。

假如说你有一个用来计算单词出现次数的 topology，你想要将单词的出现次数以 key/value 对的形式存储到数据库中。key 就是单词，value 就是这个单词出现的次数。你已经看到只是存储一个数量是不足以知道你是否已经处理过一个 batch 的。你可以通过将 value 和 txid 一起存储到数据库中。这样的话，当更新这个 count 之前，你可以先去比较数据库中存储的 txid 和现在要存储的 txid。如果一样，就跳过什么都不做，因为这个 value 之前已经被处理过了。如果不一样，就执行存储。这个逻辑可以工作的前提就是 txid 永不改变，并且 Trident 保证状态的更新是在 batch 之间严格顺序进行的。

考虑下面这个例子的运行逻辑，假定你在处理一个 txid 为 3 的包含下面 tuple 的 batch：

```
[ "man"]
[ "man"]
[ "dog"]
```

假定数据库中当前保存了下面这样的 key/value 对：

```
man => [count=3, txid=1]
dog => [count=4, txid=3]
apple => [count=10, txid=2]
```

单词“man”对应的 txid 是 1. 因为当前的 txid 是 3，你可以确定你还没有为这个 batch 中的 tuple 更新过这个单词的数量。所以你可以放心的给 count 加 2 并更新 txid 为 3. 与此同时，单词“dog”的 txid 和当前的 txid 是相同的，因此你可以跳过这次更新。此时数据库中的数据如下：

```
man => [count=5, txid=3]
dog => [count=4, txid=3]
apple => [count=10, txid=2]
```

■ Opaque transactional spouts(不透明事务 spout)

一个 opaque transactional spout 有如下特性：

每个 tuple 只在一个 batch 中被成功处理。然而，一个 tuple 在一个 batch 中被处理失败后，有可能会在另外的一个 batch 中被成功处理。

OpaqueTridentKafkaSpout 是一个拥有这种特性的 spout，并且它是容错的，即使 Kafka 的节点丢失。当 OpaqueTridentKafkaSpout 发送一个 batch 的时候，它会从上个 batch 成功结束发送的位置开始发送一个 tuple 序列。这就确保了永远没有任何一个 tuple 会被跳过或者被放在多个 batch 中被多次成功处理的情况。

使用 opaque transactional spout，再使用和 transactional spout 相同的处理方式：判断数据库中存放的 txid 和当前 txid 做对比已经不好用了。这是因为在 state 的更新过程之间，batch 可能已经变了。

你只能在数据库中存储更多的信息。除了 value 和 txid，你还需要存储之前的数值在数据库中。让我们还是用上面的例子来说明这个逻辑。假定你当前 batch 中的对应 count 是“2”，并且我们需要进行一次状态更新。而当前数据库中存储的信息如下：

```
{ value = 4, prevValue = 1, txid = 2 }
```

如果你当前的 txid 是 3，和数据库中的 txid 不同。那么就将 value 中的值设置到 prevValue 中，根据你当前的 count 增加 value 的值并更新 txid。更新后的数据库信息如下：

```
{ value = 6, prevValue = 4, txid = 3 }
```

现在再假定你的当前 txid 是 2，和数据库中存放的 txid 相同。这就说明数据库里面 value 中的值包含了之前一个和当前 txid 相同的 batch 的更新。但是上一个 batch 和当前这个 batch 可能已经完全不同了，以至于我们需要无视它。在这种情况下，你需要在 prevValue 的基础上加上当前 count 的值并将结果存放到 value 中去。数据库中的信息如下所示：

```
{ value = 3, prevValue = 1, txid = 2 }
```

因为 Trident 保证了 batch 之间的强顺序性，因此这种方法是有效的。一旦 Trident 去处理一个新的 batch，它就不会重新回到之前的任何一个 batch。并且由于 opaque transactional spout 确保在各个 batch 之间是没有共同成员的，每个 tuple 只会在一个 batch 中被成功处理，你可以安全的在之前的值上进行更新。

■ Spout 和 State 的组合

下图展示了哪些 spout 和 state 的组合能够实现有且只有一次被成功处理的语义

		State		
		Non-transactional	Transactional	Opaque transactional
Spout	Non-transactional	No	No	No
	Transactional	No	Yes	Yes
	Opaque transactional	No	No	Yes

Opaque transactional state 有着最为强大的容错性。但是这是以存储更多的信息作为代价的。Transactional states 需要存储较少的状态信息，但是仅能和 transactional spouts 协同工作。non-transactional state 所需要存储的信息最少，但是却不能实现有且只有一次被成功处理的语义。

State 和 Spout 类型的选择其实是一种在容错性和存储消耗之间的权衡，你的应用的需要会决定那种组合更适合你。

State APIs

你已经看到实现有且只有一次被执行的语义时的复杂性。Trident 这样做的好处把所有容错想过的逻辑都放在了 State 里面—作为一个用户，你并不需要自己去处理复杂的 txid，存储多余的信息到数据库中，或者是任何其他类似的事情。你只需要写如下这样简单的 code:

```
TridentTopology topology = new TridentTopology();
TridentState wordCounts =
topology.newStream("spout1", spout)
.each(new Fields("sentence"), new Split(), new Fields("word"))
.groupBy(new Fields("word"))
.persistentAggregate(MemcachedState.opaque(serverLocations), new Count(), new
Fields("count"))
.parallelismHint(6);
```

所有管理 opaque transactional state 所需的逻辑都在 MemcachedState.opaque 方法的调用中被涵盖了，除此之外，数据库的更新会自动以 batch 的形式来进行以避免多次访问数据库。State 的基本接口只包含下面两个方法：

```
public interface State {
    void beginCommit(Long txid); // can be null for things like partitionPersist
occurring off a DRPC stream
    void commit(Long txid);
}
```

当一个 State 更新开始时，以及当一个 State 更新结束时你都会被告知，并且会告诉你该次的 txid。Trident 并没有对你的 state 的工作方式有任何的假定。

假定你自己搭了一套数据库来存储用户位置信息，并且你想要在 Trident 中去访问它。则在 State 的实现中应该有用户信息的 set、get 方法：

```
public class LocationDB implements State {
    public void beginCommit(Long txid) {
    }
    public void commit(Long txid) {
    }
    public void setLocation(long userId, String location) {
        // code to access database and set location
    }
    public String getLocation(long userId) {
        // code to get location from database
    }
}
```

然后你还需要提供给 Trident 一个 StateFactory 来在 Trident 的 task 中创建你的 State 对象。LocationDB 的 StateFactory 可能会如下所示：

```
public class LocationDBFactory implements StateFactory {
    public State makeState(Map conf, int partitionIndex, int numPartitions) {
        return new LocationDB();
    }
}
```

Trident 提供了一个 QueryFunction 接口用来实现 Trident 中在一个 state source 上查询的功能。比如说，让我们写一个查询地址的操作，这个操作会查询 LocationDB 来找到用户的地址。下面以怎样在 topology 中使用该功能开始，假定这个 topology 会接受一个用户 id 作为输入数据流：

```
TridentTopology topology = new TridentTopology();
TridentState locations = topology.newStaticState(new LocationDBFactory());
topology.newStream("myspout", spout)
    .stateQuery(locations, new Fields("userid"), new QueryLocation(), new Fields("location"));
```

接下来让我们一起来看看 QueryLocation 的实现应该是什么样的：

```
public class QueryLocation extends BaseQueryFunction<LocationDB, String> {
    public List<String> batchRetrieve(LocationDB state, List<TridentTuple> inputs) {
        List<String> ret = new ArrayList();
        for(TridentTuple input: inputs) {
            ret.add(state.getLocation(input.getLong(0)));
        }
        return ret;
    }
    public void execute(TridentTuple tuple, String location, TridentCollector collector)
    {
        collector.emit(new Values(location));
    }
}
```

QueryFunction 的执行分为两部分。首先 Trident 收集了一个 batch 的 read 操作并把他们统一交给 batchRetrieve。在这个例子中，batchRetrieve 会接受到多个用户 id。batchRetrieve 应该返还一个大小和输入 tuple 数量相同的 result 列表。result 列表中的第一个元素对应着第一个输入 tuple 的结果，result 列表中的第二个元素对应着第二个输入 tuple 的结果，以此类推。

你可以看到，这段代码并没有像 Trident 那样很好的利用 batch 的优势，而是为每个输入 tuple 去查询了一次 LocationDB。所以一种更好的操作 LocationDB 方式应该是这样的：

```
public class LocationDB implements State {
    public void beginCommit(Long txid) {
    }
    public void commit(Long txid) {
    }
    public void setLocationsBulk(List<Long> userIDs, List<String> locations) {
        // set locations in bulk
    }
    public List<String> bulkGetLocations(List<Long> userIDs) {
        // get locations in bulk
    }
}
```

接着，你可以这样改写上面的 QueryLocation：

```

public class QueryLocation extends BaseQueryFunction<LocationDB, String> {
    public List<String> batchRetrieve(LocationDB state, List<TridentTuple> inputs)
    {
        List<Long> userIds = new ArrayList<Long>();
        for(TridentTuple input: inputs) {
            userIds.add(input.getLong(0));
        }
        return state.bulkGetLocations(userIds);
    }
    public void execute(TridentTuple tuple, String location, TridentCollector
collector) {
        collector.emit(new Values(location));
    }
}

```

通过有效减少访问数据库的次数，这段代码比上一个实现会高效的多。

Trident 还提供了一个 StateUpdater 来实现 Trident 中更新 state source 的功能，

如果你要更新 State，你需要使用 StateUpdater 接口，下面是一个 StateUpdater 的例子，用来将新的地址信息更新到 LocationDB 当中。

```

public class LocationUpdater extends BaseStateUpdater<LocationDB> {
    public void updateState(LocationDB state, List<TridentTuple> tuples,
TridentCollector collector) {
        List<Long> ids = new ArrayList<Long>();
        List<String> locations = new ArrayList<String>();
        for(TridentTuple t: tuples) {
            ids.add(t.getLong(0));
            locations.add(t.getString(1));
        }
        state.setLocationsBulk(ids, locations);
    }
}

```

下面列出了你应该如何在 Trident topology 中使用上面声明的 LocationUpdater：

```

TridentTopology topology = new TridentTopology();
TridentState locations =
    topology.newStream("locations", locationsSpout)
        .partitionPersist(new LocationDBFactory(), new Fields("userid",
"location"), new LocationUpdater());

```

partitionPersist 操作会更新一个 State，其内部是将 State 和一批更新的 tuple 交给 StateUpdater，由 StateUpdater 完成相应的更新操作。

在这段代码中，只是简单的从输入的 tuple 中提取出 userid 和对应的 location，并一起更新到 State 中。

partitionPersist 会返回一个 TridentState 对象来表示被这个 Trident topology 更新过的 location db。然后你就可以使用这个 state 在 topology 的任何地方进行查询操作了。

同时，你也可以看到我们传了一个 TridentCollector 给 StateUpdaters，collector 发送的 tuple 就会去往一个新的 stream。在这个例子中，我们并没有去往一个新的 stream 的需要，但是如果你在做一些事

情，比如说更新数据库中的某个 count，你可以 emit 更新的 count 到这个新的 stream。然后你可以通过调用 TridentState#newValuesStream 方法来访问这个新的 stream 来进行其他的处理。

persistentAggregate

Trident 有另外一种更新 State 的方法叫做 persistentAggregate。你在之前的 word count 例子中应该已经见过了，如下：

```
TridentTopology topology = new TridentTopology();
TridentState wordCounts =
    topology.newStream("spout1", spout)
        .each(new Fields("sentence"), new Split(), new Fields("word"))
        .groupBy(new Fields("word"))
        .persistentAggregate(new MemoryMapState.Factory(), new Count(), new
Fields("count"))
```

persistentAggregate 是在 partitionPersist 之上的另外一层抽象，它知道怎么去使用一个 Trident 聚合器来更新 State。在这个例子当中，因为这是一个 groupedstream，Trident 会期待你提供的 state 是实现了 MapState 接口的。用来进行 group 的字段会以 key 的形式存在于 State 当中，聚合后的结果会以 value 的形式存储在 State 当中。MapState 接口看上去如下所示：

```
public interface MapState<T> extends State {
    List<T> multiGet(List<List<Object>> keys);
    List<T> multiUpdate(List<List<Object>> keys, List<ValueUpdater> updaters);
    void multiPut(List<List<Object>> keys, List<T> vals);
}
```

当你在一个非 groupedstream 上面进行聚合的话，Trident 会期待你的 state 实现 Snapshottable 接口：

```
public interface Snapshottable<T> extends State {
    T get();
    T update(ValueUpdater updater);
    void set(T o);
}
```

实现 Map States

在 Trident 中实现 MapState 是非常简单的，它几乎帮你做了所有的事情。OpaqueMap, TransactionalMap, 和 NonTransactionalMap 类实现了所有相关的逻辑，包括容错的逻辑。你只需要将一个 IBackingMap 的实现提供给这些类就可以了。IBackingMap 接口看上去如下所示：

```
public interface IBackingMap<T> {
    List<T> multiGet(List<List<Object>> keys);
    void multiPut(List<List<Object>> keys, List<T> vals);
}
```

OpaqueMap's 会用 OpaqueValue 的 value 来调用 multiPut 方法，TransactionalMap's 会提供 TransactionalValue 中的 value，而 NonTransactionalMaps 只是简单的把从 Topology 获取的 object 传递给 multiPut。

Trident 还提供了一种 CachedMap 类来进行自动的 LRU cache。

另外，Trident 提供了 SnapshottableMap 类将一个 MapState 转换成一个 Snapshottable 对象。

将 **Trident** 状态持久化到数据库

由于 Storm 进行聚合的时候状态都缓存在内存中，为了保障系统的稳定性，一般将状态批量持久化到数据库如 Hbase、Mysql 等数据库中。

持久化状态到数据库中需要实现以下接口：

```
public interface IBackingMap<T> {  
    List<T> multiGet(List<List<Object>> keys);  
    void multiPut(List<List<Object>> keys, List<T> vals);  
}  
  
public interface StateFactory extends Serializable {  
    State makeState(Map conf, IMetricsContext metrics, int partitionIndex, int  
    numPartitions);  
}
```



活动 5.1：使用 **Trident Topology 结合 **StateQuery** 实现词频统计，使用 **DRPC** 服务返回统计结果**

练习问题

1. 下列哪种不是可容错的 spout?
 - a. transactionSpout
 - b. non-transactional
 - c. transactional
 - d. opaque transactional
2. 关于 transactional spout 特性描述错误的是?
 - a. 同样 txid 的 batch 一定是一样的, 当重播一个 txid 对应的 batch 时, 一定会重播和之前对应 txid 的 batch 中同样的 tuples
 - b. 各个 batch 之间是没有交集的, 每个 tuple 只能属于一个 batch
 - c. 一个 transactional spout 只能被一个 bolt 来处理
 - d. 每一个 tuple 都属于一个 batch, 无一例外
3. 可容错的 spout 描述错误的是?
 - a. 可容错的 spout 发射的数据不会丢失
 - b. Tuples 是被分成小的集合, 一组 tuple 被称为一个 batch, 被批量处理的
 - c. 每一批 tuples 被给定一个唯一 ID 作为事务 ID (txid). 当这一批 tuple 被重播时, txid 不变。
 - d. 批与批之间的状态更新时严格顺序的
4. DRPC 全称是?
 - a. Distributed Remote Procedure Call
 - b. Delay Remote Procedure Call
 - c. Distributed Remote Process Computation
 - d. Database Relational Procedure Call
5. IBackingMap 接口中定义的方法有?
 - a. multiGet
 - b. multiPut
 - c. makeState
 - d. 以上都是

小结

在本章中，您学习了：

- 可容错 spout
 - Non-transactional spout(非事务 spout)不确保每个 batch 中的 tuple 的规则（是否重叠）。
 - Trident 是以小批量（batch）的形式在处理 tuple，并且每一批都会分配一个唯一的 transaction id。
 - 不透明事务 spout: 每个 tuple 只在一个 batch 中被成功处理。
- State APIs
- persistentAggregate 是在 partitionPersist 之上的另外一层抽象, 它知道怎么去使用一个 Trident 聚合器来更新 State。
- OpaqueMap, TransactionalMap, 和 NonTransactionalMap 类实现了所有相关的逻辑，包括容错的逻辑。
- State 持久化数据库：由于 Storm 进行聚合的时候状态都缓存在内存中，为了保障系统的稳定性，一般将状态批量持久化到数据库如 Hbase、Mysql 等数据库中。
- Trident 提供批量操作数据源，可以减轻最终落地数据库的压力。合理利用 trident 可以有效提高集群处理数据的能力。

Storm 集成和管理

Storm 可以高效快速的计算一些实时数据，例如统计、数据清洗等流程，但数据最终会进入存储设备。因此保障 Storm 正常运行的同时，管理和监控 Storm 也非常的重要。本章将重点介绍 Storm 和 HBase、Storm 和 Redis 集成，以及 Storm 的监控。

目标

在本章中，您将学习：

- Storm 集成 Redis
- Storm 集成 HBase
- Storm 监控

Storm 集成 Redis

Redis 是一个 key value 类型的数据存储容器，key value 可以用 list、set、hash 等方式存储。Redis 非常快速，因为数据是存放在内存中的。Storm 集成 Redis 需要使用 Storm-redis，它使用 Jedis 作为 Redis 客户端。

■ 如何使用：

在 maven 项目里添加依赖：

```
<dependency>
    <groupId>org.apache.storm</groupId>
    <artifactId>storm-redis</artifactId>
    <version>${storm.version}</version>
</dependency>
```

■ Redis Bolt Implementations

Storm-redis 提供了一些基本的 Bolt 实现，有 RedisLookupBolt、RedisStoreBolt 以及 RedisFilterBolt。

1. RedisLookupBolt

顾名思义，RedisLookupBolt 是用于查找，通过 key 来查询 Redis 中存储的 value。

```
class WordCountRedisLookupMapper implements RedisLookupMapper {
    private RedisDataTypeDescription description;
    private final String hashKey = "wordCount";

    public WordCountRedisLookupMapper() {
        description = new RedisDataTypeDescription(
            RedisDataTypeDescription.RedisDataType.HASH, hashKey);
    }

    @Override
    public List<Values> toTuple(ITuple input, Object value) {
        String member = getKeyFromTuple(input);
        List<Values> values = Lists.newArrayList();
        values.add(new Values(member, value));
        return values;
    }

    @Override
```

```

public void declareOutputFields(OutputFieldsDeclarer declarer) {
    declarer.declare(new Fields("wordName", "count"));
}
@Override
public RedisDataTypeDescription getDataTypeDescription() {
    return description;
}

@Override
public String getKeyFromTuple(ITuple tuple) {
    return tuple.getStringByField("word");
}

@Override
public String getValueFromTuple(ITuple tuple) {
    return null;
}
}

```

用法：

```

JedisPoolConfig poolConfig = new JedisPoolConfig.Builder()
    .setHost(host).setPort(port).build();
RedisFilterMapper filterMapper = new BlacklistWordFilterMapper();
RedisFilterBolt filterBolt = new RedisFilterBolt(poolConfig,
    filterMapper);

```

2. RedisStoreBolt

顾名思义，RedisStoreBolt 是用于存储，以 key/value 的形式向 Redis 中保存数据。

```

class WordCountStoreMapper implements RedisStoreMapper {
    private RedisDataTypeDescription description;
    private final String hashKey = "wordCount";

    public WordCountStoreMapper() {
        description = new RedisDataTypeDescription(
            RedisDataTypeDescription.RedisDataType.HASH, hashKey);
    }

    @Override
    public RedisDataTypeDescription getDataTypeDescription() {
        return description;
    }

    @Override
    public String getKeyFromTuple(ITuple tuple) {
        return tuple.getStringByField("word");
    }

    @Override
    public String getValueFromTuple(ITuple tuple) {
        return tuple.getStringByField("count");
    }
}

```

用法:

```
JedisPoolConfig poolConfig = new JedisPoolConfig.Builder()
    .setHost(host).setPort(port).build();
RedisStoreMapper storeMapper = new WordCountStoreMapper();
RedisStoreBolt storeBolt = new RedisStoreBolt(poolConfig, storeMapper);
```

3. RedisFilterBolt

顾名思义，RedisFilterBolt 是用于过滤，用于从 tuple 中过滤掉 Redis 中不包含其 key 或者 field 的数据。

```
class BlacklistWordFilterMapper implements RedisFilterMapper {
    private RedisDataTypeDescription description;
    private final String setKey = "blacklist";

    public BlacklistWordFilterMapper() {
        description = new RedisDataTypeDescription(
            RedisDataTypeDescription.RedisDataType.SET, setKey);
    }

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

    @Override
    public RedisDataTypeDescription getDataTypeDescription() {
        return description;
    }

    @Override
    public String getKeyFromTuple(ITuple tuple) {
        return tuple.getStringByField("word");
    }

    @Override
    public String getValueFromTuple(ITuple tuple) {
        return null;
    }
}
```

用法:

```
JedisPoolConfig poolConfig = new JedisPoolConfig.Builder()
    .setHost(host).setPort(port).build();
RedisFilterMapper filterMapper = new BlacklistWordFilterMapper();
RedisFilterBolt filterBolt = new RedisFilterBolt(poolConfig,
    filterMapper);
```

Storm 集成 HBase

Storm 是一个实时数据处理系统，然而，在大多数情况下，您需要将处理过的数据存储在数据库中，您可以使用存储的数据进行进一步的分析，并可以执行分析数据存储查询。接下来将讲解如何存储数据进入 Hbase。

■ 如何使用

在 maven 项目里添加依赖:

```
<dependency>
    <groupId>org.apache.storm</groupId>
    <artifactId>storm-hbase</artifactId>
    <version>${storm.version}</version>
</dependency>
```

其中负责和 HBase 进行交互的 API 主要来自接口

org.apache.storm.hbase.bolt.mapper.HBaseMapper:

```
HBaseMapper extends Serializable {
    byte[] rowKey(Tuple tuple);
    ColumnList columns(Tuple tuple);
}
```

rowKey()方法很简单:给定一个 Storm 元组 (tuple) , 返回一个表示行键 (row key) 的字节数组。columns()方法定义了要写入 HBase 行的内容。ColumnList 类允许您添加标准 HBase 列和 HBase 计数器列 (counter columns) 。要添加标准列, 请使用 addColumn()方法:

■ SimpleHBaseMapper

Storm-hbase 包括一个名为 SimpleHBaseMapper 的通用 HBaseMapper 实现, 它可以 Storm 元组映射到常规 HBase 列和计数器列(添加新值的同时会累加旧值)。要使用 SimpleHBaseMapper, 只需告诉它要映射哪些字段到哪些类型的列。下面的代码创建了一个 SimpleHBaseMapper 实例:

1. 使用单词元组值作为行键。
2. 为元组字段单词添加标准 HBase 列。
3. 为元组字段计数添加 HBase 计数器列。
4. 将值写入 cf 列族。

```
SimpleHBaseMapper mapper = new SimpleHBaseMapper()
    .withRowKeyField("word")
    .withColumnFields(new Fields("word"))
    .withCounterFields(new Fields("count"))
    .withColumnFamily("cf");
```

■ HBaseBolt

要构造一个 HBaseBolt，需要在其构造函数中传入一个表的名称以及一个 HBaseMapper 的实现：

```
HBaseBolt hbase = new HBaseBolt("WordCount", mapper);
```

以 word count 的例子讲解如何实例化 HBaseBolt

```
// storm config
Config config = new Config();
// hbase conf
Map<String, Object> hbConf = new HashMap<>();
hbConf.put("hbase.rootdir", "hdfs://192.168.186.100:9000/hbase");
hbConf.put("hbase.zookeeper.quorum", "192.168.186.100:2181");
config.put("hbase.conf", hbConf);
SimpleHBaseMapper mapper = new SimpleHBaseMapper()
    .withRowKeyField("word")
    .withColumnFields(new Fields("word"))
    .withCounterFields(new Fields("count"))
    .withColumnFamily("cf");
HBaseBolt hBaseBolt = newHBaseBolt("wc", mapper).withConfigKey("hbase.conf");
```

执行代码之前，需要在 HBase 中提前创建 1 个 wc 表，使用以下命令：

```
Hbase>create 'wc', 'cf'
```



活动 6.1：Storm 集成 Redis，并将 Storm 中的数据持久化到 Redis 中存储



活动 6.2：Storm 集成 HBase，并将 Storm 中的数据持久化到 HBase 中存储

Storm 监控

本章将通过以下专题来介绍 Storm 集群监控

- 启动 Storm UI
- 使用 Storm UI 来监控 topology

这章节将展示如何启动 Storm UI，假设你已经启动运行了一个 Storm 集群，通过如下命令启动

```
> bin/storm ui
```

默认情况下 Storm UI 启动后监听 8080 端口，我们可以通过访问 <http://nimbus-node:8080> 页面去浏览 Storm UI：

The screenshot shows the Storm UI dashboard with the following sections:

- Cluster Summary:** Displays cluster statistics:

Version	Nimbus uptime	Supervisors	Used slots	Free slots	Total slots	Executors	Tasks
0.9.0.1	11m 33s	2	0	4	4	0	0
- Topology summary:** Displays topology details:

Name	Id	Status	Uptime	Num workers	Num executors	Num tasks
Topology 1	1	ACTIVE	11m 33s	1	1	1
- Supervisor summary:** Displays supervisor details:

Id	Host	Uptime	Slots	Used slots
0xaac36e9-5904-41a5-a9ea-a95cc1ef890	Supervisor 1	11m 14s	4	0
f70c269c-7b0b-4154-a8c3-f95bbbea4a8c	Supervisor 2	11m 30s	4	0
- Nimbus Configuration:** Displays configuration settings:

Key	Value
dev.zookeeper.path	/tmp/dev-storm-zookeeper
drpc.childopts	-Xmx768m
drpc.invocations.port	3773
drpc.port	3772
drpc.queue.size	128

■ Cluster Summary

- Nimbus uptime: nimbus 的启动时间
- Supervisors: storm 集群中 supervisor 的数目
- Used slots: 使用了的 slots 数
- Free slots: 剩余的 slots 数
- Total slots: 总的 slots 数
- Running tasks: 运行的任务数

■ Topology summary

- Name: topology name
- Id: topology id (由 storm 生成)
- Status: topology 的状态，包括(ACTIVE, INACTIVE, KILLED, REBALANCING)
- Uptime: topology 运行的时间

Num workers: 运行的 workers 数
Num tasks: 运行的 task 数

- Supervisor summary
 - Host: supervisor(主机)的主机名
 - Uptime: supervisor 启动的时间
 - Slots: supervisor 的端口数
 - Used slots: 使用的端口数

UI 方式监控 Storm 集群

这章节包含了我们如何利用 Storm UI 来监控集群。监控的作用是用来检测在运行在集群中各种组件的健康情况：

- **Cluster Summary:** 这部分显示 Strom 的版本, nimbus 在线时间、空闲的 worker slots 总数、已使用的 workder slots 总数, 等待。当往集群提交一个 topology 的时候, 用户首先需要确认 Free slots 栏数量不能为 0, 否则不能获取任何 worker 去处理这个 topology, 直到队列中有空闲的 worker。
- **Nimbus Configuration:** 这部分显示 Nimbus 节点的配置信息。
- **Supervisor summary:** 这个部分用一个列表显示 supervisor 节点在集群中的运行情况, 包含 ID, Host, 在线时间、slots 和已经使用 slots 栏。
- **Topology summary:** 这个部分用一个列表显示在集群中 topology 的数量, 包含 ID,topology 指定的 worker 数量、executor 数量、task 数量、在线时间等

远程执行一个 Storm 命令如下：

```
bin/storm jar $STORM_PROJECT_HOME/target/storm-example-0.0.1-SNAPSHOT-jar-with-dependencies.jar com.learningstorm.storm_example.  
LearningStormSingleNodeTopology LearningStormClusterTopology
```

提交到集群后可以在 Topology Summary 中查看集群的状态（ACTIVE、KILLED、INACTIVE），包含 topology 名称、唯一的 ID、topology 状态、在线时长、worker 数量等待。

当点开 LearningStormClusterTopology 链接后，出现如下界面：

Topology summary						
Name	Id	Status	Uptime	Num workers	Num executors	Num tasks
LearningStormClusterTopology	LearningStormClusterTopology-1-1a04196814	ACTIVE	1m 36s	3	9	9

Topology actions						
<input type="button" value="Activate"/> <input type="button" value="Deactivate"/> <input type="button" value="Rebalance"/> <input type="button" value="Kill"/>						

Topology stats						
Window	Emitted	Transferred	Complete latency (ms)	Acked	Failed	
10m 0s	16250600	16250600	0.000	0	0	
3h 0m 0s	16250600	16250600	0.000	0	0	
1d 0h 0m 0s	16250600	16250600	0.000	0	0	
All time	16250600	16250600	0.000	0	0	

Spouts (All time)								
Id	Executors	Tasks	Emitted	Transferred	Complete latency (ms)	Acked	Failed	Last error
LearningStormSpout	2	2	16250600	16250600	0.000	0	0	

Bolts (All time)											
Id	Executors	Tasks	Emitted	Transferred	Capacity (last 10m)	Execute latency (ms)	Executed	Process latency (ms)	Acked	Failed	Last error
LearningStormBolt	4	4	0	0	0.057	0.001	12178960	0.000	12178900	0	

- Topology actions: 这部分允许我们去通过 Storm UI 直接去 activate、deactivate、rebalance、kill 这个 topology。
- topology status:
 - **emitted:** emitted tuple 数
 - **transferred:** transferred tuple 数, 说下与 emitted 的区别: 如果一个 task, emitted 一个 tuple 到 2 个 task 中, 则 transferred tuple 数是 emitted tuple 数的两倍。
 - **complete latency:** spout emitting 一个 tuple 到 spout ack 这个 tuple 的平均时间
 - **acked:** ack tuple 数
 - **failed:** 失败的 tuple 数
- Spouts (All time): 这部分统计在一个 topology 里面所有的 spouts
 - **Executors:** 这栏显示 executors 的实际数量
 - **Tasks :** 显示所有给定 Tasks 的数量
 - **Emitted:** 显示已经被发送的所有记录
 - **Port:** 运行这个 topology 的端口
 - **Transferred:** 显示这个 topology 已经传输的所有记录数
 - **Complete latency (ms):** spout emitting 一个 tuple 到 spout ack 这个 tuple 的平均时间
- Bolts (All time): 这部分统计在一个 topology 里面所有的 bolt
 - **Executors:** 这栏显示 executors 的实际数量
 - **Tasks :** 显示所有给定 Tasks 的数量
 - **Emitted:** 显示已经被发送的所有记录
 - **Port:** 运行这个 topology 的端口
 - **Transferred:** 显示这个 topology 已经传输的所有记录数
 - **Complete latency (ms):** 一个 tuple 到 bolt ack 这个 tuple 的平均时间

使用 API 监控 Storm

Storm 提供了 UI 方式来监控集群的运行负载情况，同时 Storm 提供了 Thrift Java API 方式来管理 Storm 集群，提供了以下方式：

- 收集 nimbus 的 Configuration
- 收集 supervisor 的统计信息
- 收集 topology 的统计信息
- 收集指定 topology 中的 spout 信息
- 收集指定 topology 中的 bolt 信息
- 停止指定 topology

为完成上述内容，首先需要创建一个工具类，来完成和 nimbus thrift server 连接，并且返回 Nimbus 的 Client 信息：

```
public class ThriftClient {  
    // IP of the Storm UI node  
    private static final String STORM_UI_NODE = "127.0.0.1";  
    public Client getClient() {  
        // Set the IP and port of thrift server.  
        // By default, the thrift server start on port 6627  
        TSocket socket = new TSocket(STORM_UI_NODE, 6627);  
        TTransport tFramedTransport = new TTransport(socket);  
        TBinaryProtocol tBinaryProtocol = new TBinaryProtocol(tFramedTransport);  
        Client client = new Client(tBinaryProtocol);  
        try {  
            // Open the connection with thrift client.  
            tFramedTransport.open();  
        } catch (Exception exception) {  
            throw new RuntimeException("Error occurred while making connection with  
nimbus thrift server");  
        }  
        // return the Nimbus Thrift client.  
        return client;  
    }  
}
```

- 收集 Nimbus 的 Configuration

```
public class NimbusConfiguration {  
    public void printNimbusStats() {  
        try {  
            ThriftClient thriftClient = new ThriftClient();  
            Client client = thriftClient.getClient();  
            String nimbusConiguration = client.getNimbusConf();  
            System.out.println ("*****");  
            System.out.println ("Nimbus Configuration : "+nimbusConiguration);  
            System.out.println ("*****");  
        } catch (Exception exception) {  
            throw new RuntimeException("Error occurred while fetching the Nimbus  
Configuration");  
        }  
    }  
}
```

```

        statistics : ");
    }
}
public static void main(String[] args) {
    new NimbusConfiguration().printNimbusStats();
}
}

```

■ 收集 supervisor 的统计信息

```

public class SupervisorStatistics {
    public void printSupervisorStatistics() {
        try {
            ThriftClient thriftClient = new ThriftClient();
            Client client = thriftClient.getClient();
            // Get the cluster information.
            ClusterSummary clusterSummary = client.getClusterInfo();
            // Get the SupervisorSummary iterator
            Iterator<SupervisorSummary> supervisorsIterator =
                clusterSummary.get_supervisors_iterator();
            while (supervisorsIterator.hasNext()) {
                // Print the information of supervisor node
                SupervisorSummary supervisorSummary = (SupervisorSummary)
                    supervisorsIterator.next();
                System.out.println ("*****");
                System.out.println ("Supervisor Host IP : "+supervisorSummary.get_host());
                System.out.println("Number of used workers :
                    "+supervisorSummary.get_num_used_workers());
                System.out.println("Number of workers :
                    "+supervisorSummary.get_num_workers());
                System.out.println("Supervisor ID :
                    "+supervisorSummary.get_supervisor_id());
                System.out.println("Supervisor uptime in seconds :
                    "+supervisorSummary.get_uptime_secs());
                System.out.println ("*****");
            }
        } catch (Exception e) {
            throw new RuntimeException("Error occurred while getting cluster info : ");
        }
    }
}

```

■ 收集 topology 的统计信息

```

public class TopologyStatistics {
    public void printTopologyStatistics() {
        try {
            ThriftClient thriftClient = new ThriftClient();
            // Get the thrift client
            Client client = thriftClient.getClient();
            // Get the cluster info
            ClusterSummary clusterSummary = client.getClusterInfo();
            // Get the interator over TopologySummary class
            Iterator<TopologySummary> topologiesIterator =

```

```

clusterSummary.get_topologies_iterator();
while (topologiesIterator.hasNext()) {
TopologySummary topologySummary = topologiesIterator.next();
System.out.println ("*****");
System.out.println("ID of topology: " + topologySummary.get_id());
System.out.println("Name of topology: " + topologySummary.get_name());
System.out.println("Number of Executors: " +
topologySummary.get_num_executors());
System.out.println("Number of Tasks: " + topologySummary.get_num_tasks());
System.out.println("Number of Workers: " +
topologySummary.get_num_workers());
System.out.println("Status of topology: " + topologySummary.get_status());
System.out.println("Topology uptime in seconds: " +
topologySummary.get_uptime_secs());
System.out.println ("*****");
}
} catch (Exception exception) {
throw new RuntimeException("Error occurred while fetching the topologies
information");
}
}
}
}

```

■ 收集指定 topology 中的 spout 信息

```

public class SpoutStatistics {
private static final String DEFAULT = "default";
private static final String ALL_TIME = ":all-time";
public void printSpoutStatistics(String topologyId) {
try {
ThriftClient thriftClient = new ThriftClient();
// Get the nimbus thrift client
Client client = thriftClient.getClient();
// Get the information of given topology
TopologyInfo topologyInfo = client.getTopologyInfo(topologyId);
Iterator<ExecutorSummary> executorSummaryIterator =
topologyInfo.get_executors_iterator();
while (executorSummaryIterator.hasNext()) {
ExecutorSummary executorSummary = executorSummaryIterator.next();
ExecutorStats executorStats = executorSummary.get_stats();
if(executorStats !=null) {
ExecutorSpecificStats executorSpecificStats = executorStats.get_specific();
String componentId = executorSummary.get_component_id();

if (executorSpecificStats.is_set_spout()) {
SpoutStats spoutStats = executorSpecificStats.get_spout();
System.out.println ("*****");
System.out.println ("Component ID of Spout:- " + componentId);
System.out.println("Transferred:- " +
getAllTimeStat(executorStats.get_transferred(), ALL_TIME));
System.out.println("Total tuples emitted:- " +
getAllTimeStat(executorStats.get_emitted(), ALL_TIME));
System.out.println("Acked: " + getAllTimeStat(spoutStats.get_acked(),

```

```

        ALL_TIME));
System.out.println("Failed: " + getAllTimeStat(spoutStats.get_failed(),
ALL_TIME));
System.out.println ("*****");
}
}
}
}catch (Exception exception) {
throw new RuntimeException("Error occurred while fetching the spout
information : "+exception);
}
}

private static Long getAllTimeStat(Map<String, Map<String, Long>> map,
String statName) {
if (map != null) {
Long statValue = null;
Map<String, Long> tempMap = map.get(statName);
statValue = tempMap.get(DEFAULT);
return statValue;
}
return 0L;
}
public static void main(String[] args) {
new SpoutStatistics().printSpoutStatistics ("LearningStormClusterTopology-
1-1393847956");
}
}

```

■ 收集指定 topology 中的 bolt 信息

```

public class BoltStatistics {
private static final String DEFAULT = "default";
private static final String ALL_TIME = ":all-time";
public void printBoltStatistics(String topologyId) {
try {
ThriftClient thriftClient = new ThriftClient();
// Get the Nimbus thrift server client
Client client = thriftClient.getClient();
// Get the information of given topology
TopologyInfo topologyInfo = client.getTopologyInfo(topologyId);
Iterator<ExecutorSummary> executorSummaryIterator =
topologyInfo.get_executors_iterator();
while (executorSummaryIterator.hasNext()) {
// get the executor
ExecutorSummary executorSummary = executorSummaryIterator.next();
ExecutorStats executorStats = executorSummary.get_stats();
if (executorStats != null) {
ExecutorSpecificStats executorSpecificStats =
executorStats.get_specific();
String componentId = executorSummary.get_component_id();
if (executorSpecificStats.is_set_bolt()) {
BoltStats boltStats = executorSpecificStats.get_bolt();
System.out.println ("*****");
}
}
}
}
}

```

```

System.out.println ("Component ID of Bolt " + componentId);
System.out.println("Transferred: " +
getAllTimeStat(executorStats.get_transferred(), ALL_TIME));
System.out.println("Emitted: " +
getAllTimeStat(executorStats.get_emitted(), ALL_TIME));
System.out.println("Acked: " + getBoltStats(boltStats.get_acked(),
ALL_TIME));
System.out.println("Failed: " + getBoltStats( boltStats.get_failed(),
ALL_TIME));
System.out.println("Executed: " + getBoltStats(boltStats.get_executed(),
ALL_TIME));
System.out.println ("*****");
}
}

} } catch (Exception exception) {
throw new RuntimeException("Error occurred while fetching the bolt
information :" +exception);
}
}

private static Long getAllTimeStat(Map<String, Map<String, Long>> map,
String statName) {
if (map != null) {
Long statValue = null;
Map<String, Long> tempMap = map.get(statName);
statValue = tempMap.get(DEFAULT);
return statValue;
}
return 0L;
}

public static Long getBoltStats(Map<String, Map<GlobalStreamId, Long>> map,
String statName) {
if (map != null) {
Long statValue = null;
Map<GlobalStreamId, Long> tempMap = map.get(statName);
Set<GlobalStreamId> key = tempMap.keySet();
if (key.size() > 0) {
Iterator<GlobalStreamId> iterator = key.iterator();
statValue = tempMap.get(iterator.next());
}
return statValue;
}
return 0L;
}

public static void main(String[] args) {
new BoltStatistics(). printBoltStatistics ("LearningStormClusterTopology-
1-1393847956");
}
}

```

■ 停止指定 topology

```
public class KillTopology {
    public void kill(String topologyId) {
        try {
            ThriftClient thriftClient = new ThriftClient();
            // Get the nimbus thrift client
            Client client = thriftClient.getClient();
            // kill the given topology
            client.killTopology(topologyId);
        } catch (Exception exception) {
            throw new RuntimeException("Error occurred while killing the topology : " + exception);
        }
    }

    public static void main(String[] args) {
        new KillTopology().kill("topologyId");
    }
}
```



使用 JMX 监控 Storm

我们已经学习了如何使用 Storm UI 或 thrift API 监视 Storm 集群。本节将解释如何使用 JMX 监视 Storm 集群。JMX 是一组用于管理和监视在 JVM 中运行的应用程序的规范。我们可以收集或显示 Storm 指标，比如堆大小、非堆大小、线程数量、加载的类数量、堆和非堆内存以及虚拟机参数，并在 JMX 控制台上管理对象。以下是使用 JMX 监视 Storm 集群需要执行的步骤：

首先我们需要在每个 supervisor 节点上的 storm.yaml 文件中开启 JMX，添加下面几行：

```
supervisor.childopts:  
  -verbose:gc  
  -XX:+PrintGCTimeStamps  
  -XX:+PrintGCDetails  
  -Dcom.sun.management.jmxremote  
  -Dcom.sun.management.jmxremote.ssl=false  
  -Dcom.sun.management.jmxremote.authenticate=false  
  -Dcom.sun.management.jmxremote.port=12346
```

这里的 12346 是用来通过 JMX 连接到 supervisor 上的 Java 虚拟机(JVM) 并获取指标哦（metrics）的端口号。.

然后在每个 Nimbus 节点上的 storm.yaml 文件中开启 JMX，添加下面几行：:

```
nimbus.childopts:  
  -verbose:gc  
  -XX:+PrintGCTimeStamps  
  -XX:+PrintGCDetails  
  -Dcom.sun.management.jmxremote  
  -Dcom.sun.management.jmxremote.ssl=false  
  -Dcom.sun.management.jmxremote.authenticate=false  
  -Dcom.sun.management.jmxremote.port=12345
```

这里的 12345 是用来通过 JMX 连接到 Nimbus 的 JVM 上收集指标信息（metrics）的端口号。.

如果还需要收集 supervisor 节点上所有 worker 进程的指标信息（metrics），可以在 supervisor 节点的 storm.yaml 文件中添加：:

```
worker.childopts:  
  -verbose:gc  
  -XX:+PrintGCTimeStamps  
  -XX:+PrintGCDetails  
  -Dcom.sun.management.jmxremote  
  -Dcom.sun.management.jmxremote.ssl=false  
  -Dcom.sun.management.jmxremote.authenticate=false  
  -Dcom.sun.management.jmxremote.port=2%ID%
```

在这里，%ID%表示 worker 进程的端口号。如果工作进程的端口是 6700，那么它的 JVM 指标将发布在端口 26700 上(2%ID%)。现在可在任何安装了 Java 的机器上运行以下命令来启动 JConsole：

```
cd $JAVA_HOME  
./bin/jconsole
```

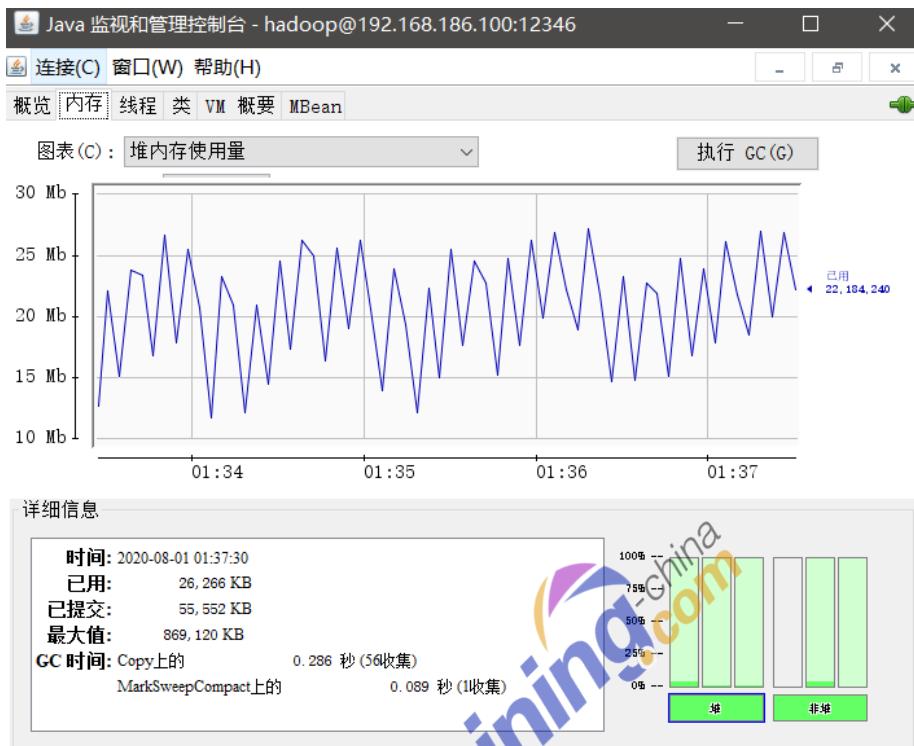
下面 JMX 连接页面的屏幕截图显示了如何使用 JConsole 连接 JMX 端口：



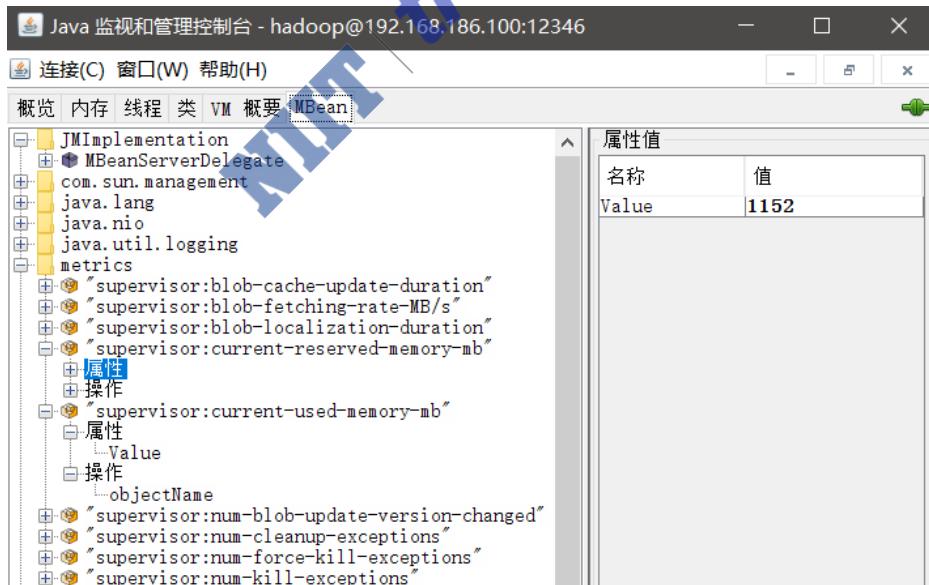
JConsole

如果您想查看远程节点的 JMX 控制台，那么您需要填入远程节点的实际的 IP 地址，而不是 192.168.186.100。

现在，单击 Connect 按钮查看监控器节点的指标。下面的屏幕截图显示了 Storm 监控器节点在 JMX 控制台上的指标：



内存监控界面:



练习问题

1. 编写 Storm 和 HBase 或者 Redis 集成，一般在哪里初始化连接？
 - a. init 方法
 - b. declareOutputFields
 - c. prepare
 - d. execute
2. 如果 Storm 插入 HBase 时速度比较慢，一般会导致什么情况？
 - a. spout 发射数据比较慢
 - b. 数据都在队列积压
 - c. 会导致 OOM
 - d. zookeeper 中数据大量积压
3. Storm UI 默认端口是？
 - a. 80
 - b. 8080
 - c. 88
 - d. 9090
4. 使用 Storm UI 监控集群运行时会显示什么信息？
 - a. Cluster 概述
 - b. Nimbus 配置
 - c. Supervisor 概况
 - d. 以上都是
5. SimpleHBaseMapper 中定义的方法有？
 - a. withRowKeyField("word") and withColumnFamily("cf");
 - b. withColumnFields(new Fields("word"));
 - c. withCounterFields(new Fields("count"));
 - d. 以上都是

小结

本章您学习了：

- Storm 集成 Redis
- Storm 集成 Hbase
- 使用 Storm UI 监控 Storm
- 使用 API 监控 Storm
- 使用 JMX 监控 Storm
- Storm 作为实时流式计算的热门框架，在于高效、持续的运算。一般在实际应用中使用 Redis 来做 bolt 的查询服务，使用 HBase 作为落地存储。项目中大多都是将三者结合在一起使用。

