

# Storm 1.1.0 中文文档

来源: ApacheCN

## NOTE (注意)

在最新版本中, class packages 已经从 "backtype.storm" 改变成 "org.apache.storm" 了, 所以使用旧版本编译的 topology 代码不会像在 Storm 1.0.0 上那样运行了. 通过以下配置提供向后的兼容性

```
client.jartransformer.class: "org.apache.storm.hack.StormShadeTransformer"
```

如果要运行使用较旧版本 Storm 编译的代码, 则需要在 Storm 安装中添加上述配置. 该配置应该添加到您用于提交 topologies (拓扑) 的机器中.

更多细节, 请参阅 <https://issues.apache.org/jira/browse/STORM-1202>.

## Storm 基础

- Javadoc
- 概念
- 调度器
- 配置
- 保证消息处理
- Daemon (守护进程) 容错
- 命令行 client (客户端)
- REST API
- 理解 Storm topology 的 parallelism (并行度)
- FAQ

## Layers on Top of Storm

### Storm Trident

Trident 是 Storm 的另一个 interface (接口). 它提供了 exactly-once (仅且一次) 处理, "transactional (事务性的)" datastore persistence (数据存储持久化), 以及一些常见的

# Storm 1.1.0 中文文档

来源: [ApacheCN](#)

## NOTE (注意)

在最新版本中, class packages 已经从 "backtype.storm" 改变成 "org.apache.storm" 了, 所以使用旧版本编译的 topology 代码不会像在 Storm 1.0.0 上那样运行了. 通过以下配置提供向后的兼容性

```
client.jartransformer.class: "org.apache.storm.hack.StormShadeTransformer"
```

如果要运行使用较旧版本 Storm 编译的代码, 则需要在 Storm 安装中添加上述配置. 该配置应该添加到您用于提交 topologies (拓扑) 的机器中.

更多细节, 请参阅 <https://issues.apache.org/jira/browse/STORM-1202>.

## Storm 基础

- [Javadoc](#)
- 概念
- 调度器
- 配置
- 保证消息处理
- Daemon (守护进程) 容错
- 命令行 client (客户端)
- REST API
- 理解 Storm topology 的 parallelism (并行度)
- FAQ

## Layers on Top of Storm

### Storm Trident

Trident 是 Storm 的另一个 interface (接口). 它提供了 exactly-once (仅且一次) 处理, "transactional (事务性的)" datastore persistence (数据存储持久化), 以及一些常见的 stream analytics operations (流式分析操作).

- [Trident 教程 -- 基础的概念和预排工作](#)

- [Trident API 概述](#) -- 针对 transforming (转换) 和 orchestrating 数据的操作
- [Trident State \(状态\)](#) -- exactly-once (仅且一次) 处理以及 fast (快速的), persistent aggregation (持久化的聚合)
- [Trident spouts](#) -- transactional (事务性的) 和 non-transactional (非事务性的) 数据引入
- [Trident RAS API](#) -- 与 Trident 一起使用 Resource Aware Scheduler .

## Storm SQL

该 Storm SQL 的集成可以让用户在 Storm 的 streaming data (流式数据) 上来运行 SQL 查询.

NOTE (注意) : Storm SQL 是一个 `experimental` (实验性的) 功能, 所以 Storm SQL 的结构和所支持的功能在以后可能会发生变化. 但是小的变化不会影响用户体验. 在引入 UX 更改时, 我们会及时通知用户.

- [Storm SQL 概述](#)
- [Storm SQL 示例](#)
- [Storm SQL 文献](#)
- [Storm SQL 结构](#)

## Flux

- [Flux Data Driven Topology Builder](#)

## Storm 安装和部署

- 安装一个 Storm 集群
- [Local mode \(本地模式\)](#)
- 问题排查
- 在生产 cluster (集群) 上运行 topologies (拓扑)
- 构建 Storm with Maven
- 安装 Secure (安全的) Cluster (集群)
- CGroup 的实施
- Pacemaker 针对大集群减低在 zookeeper 上的负载
- [Resource Aware Scheduler \(资源意识调度器\)](#)

- [Daemon Metrics/Monitoring](#) (守护进程的度量/监控)
- [Windows 平台的用户指南](#)

## Storm 中级

- [Serialization](#) (序列化)
- [Common patterns](#) (常见模式)
- [Clojure DSL](#)
- 与 Storm 一起使用非 JVM 的语言
- 分布式的 RPC
- [Transactional topologies](#) (事务性的拓扑)
- [Hooks](#) (钩子)
- [Metrics](#) (度量)
- [State Checkpointing](#)
- [Windowing](#) (窗口操作)
- [Joining Streams](#)
- [Blobstore\(Distcahce\)](#)

## Storm 调试

- [Dynamic Log Level Settings](#)
- [Searching Worker Logs](#)
- [Worker Profiling](#)
- [Event Logging](#)

## Storm 与外部系统, 以及其它库的集成

- [Apache Kafka 集成, 新的 Kafka Consumer](#) (消费者) 集成
- [Apache HBase 集成](#)
- [Apache HDFS 集成](#)
- [Apache Hive 集成](#)
- [Apache Solr 集成](#)
- [Apache Cassandra 集成](#)
- [JDBC 集成](#)
- [JMS 集成](#)

- Redis 集成
- Event Hubs 集成
- Elasticsearch 集成
- MQTT 集成
- Mongodb 集成
- OpenTSDB 集成
- Kinesis 集成
- Druid 集成
- Kestrel 集成

## Container, Resource Management System Integration

- YARN 集成, 通过 Slider 集成 YARN
- Mesos 集成
- Docker 集成
- Kubernetes 集成

## Storm 高级

- 为 Storm 定义非 JVM 语言的 DSL
- 多语言协议 (如何为其它语言提供支持)
- 实现文档

# Storm 基础

# 概念

本页列出了Storm 的主要概念, 以及可以获取到更多信息的资源链接, 概念如下:

1. Topologies (拓扑)
2. Streams (流)
3. Spouts
4. Bolts
5. Stream groupings (流分组)
6. Reliability (可靠性)
7. Tasks
8. Workers

## Topologies (拓扑)

实时应用程序的逻辑被封装在 Storm topology (拓扑) 中. Storm topology (拓扑) 类似于 MapReduce 作业. 两者之间关键的区别是 MapReduce 作业最终会完成, 而 topology (拓扑) 任务会永远运行 (除非 kill 掉它). 一个拓扑是 Spout 和 Bolt 通过 stream groupings 连接起来的有向无环图. 这些概念会在下面的段落中具体描述.

相关资料:

- [TopologyBuilder](#): Java中使用这个类来构建 topology (拓扑)
- [如何在生产集群上运行 topologies \(拓扑\)](#)
- [如何使用 local 模式](#): 学习如何用本地模式开发和测试 topology (拓扑)

## Streams (流)

stream 是 Storm 中的核心概念. 一个 stream 是一个无界的、以分布式方式并行创建和处理的 Tuple 序列. stream 以一个 schema 来定义, 这个 schema 用来命名 stream tuple (元组) 中的字段. 默认情况下 Tuple 可以包含 integers, longs, shorts, bytes, strings, doubles, floats, booleans, and byte arrays 等数据类型. 你也可以定义自己的 serializers, 以至于可以在 Tuple 中使用自定义的类型.

每一个流在声明的时候会赋予一个 ID. 由于只包含一个 stream 的 Spout 和 Bolt 比较常见, [OutputFieldsDeclarer](#) 有更方便的方法可以定义一个单一的 stream 而不用指定ID. 这个 stream 被赋予一个默认的 ID, "default".

相关资料:

- [Tuple](#): stream 由一系列连续的 Tuple 组成
- [OutputFieldsDeclarer](#): 用于声明 streams 和它的 schemas.
- [Serialization](#): Storm Tuple 的动态类型, 和自定义 serializations 的相关信息

## Spouts

Spout 是一个 topology (拓扑) 中 streams 的源头. 通常 Spout 会从外部数据源读取 Tuple, 然后把他们发送到拓扑中 (如 Kestrel 队列, 或者 Twitter API). Spout 可以是可靠的或不可靠的. 可靠的 Spout 在 Storm 处理失败的时候能够重放 Tuple, 不可靠的 Spout 一旦把一个 Tuple 发送出去就撒手不管了.

Spout 可以发送多个流. 可以使用 [OutputFieldsDeclarer](#) 的 `declareStream` 方法定义多个流, 在 [SpoutOutputCollector](#) 对象的 `emit` 方法中指定要发送到的 stream .

Spout 中的最主要的方法是 `nextTuple`. `nextTuple` 要么向 topology (拓扑) 中发送一个新的 Tuple, 要么在没有 Tuple 需要发送的情况下直接返回. 对于任何 Spout 实现, `nextTuple` 方法都必须非阻塞的, 因为 Storm 在一个线程中调用所有的 Spout 方法.

Spout 的另外几个重要的方法是 `ack` 和 `fail`. 这些方法在 Storm 检测到 Spout 发送出去的 Tuple 被成功处理或者处理失败的时候调用. `ack` 和 `fail` 只会在可靠的 Spout 中调用. 更多相关信息, 请参见 [the Javadoc](#).

相关资料:

- [IRichSpout](#): 创建 Spout 时必须实现的接口
- [Guaranteeing message processing](#)

## Bolts

拓扑中所有的业务处理都在 Bolts 中完成. Bolt 可以做很多事情, 过滤, 函数, 聚合, 关联, 与数据库交互等.

Bolt 可以做简单 stream 转换. 复杂的 stream 转换一般需要多个步骤, 因此也就要多个 Bolt 协同工作. 如, 转换一个 tweets stream 为一个 trending images stream 需要两个步骤: 一个 Bolt 做每个图片被收藏 的滚动计数, 同时一个或者多个 Bolt 输出被收藏 Top X 的图片 (你可以使用更具弹性的方式处理这个 stream 转换, 用3个 Bolt 而不是先前的2个 Bolt ).

Bolt 可以发送多个 stream. 可以使用 [OutputFieldsDeclarer](#) 的 `declareStream` 方法定义多个 streams, 并且在使用 [OutputCollector](#) `emit` 方法的时候指定要发送的 stream.

当你声明一个 Bolt 的 input stream, 你总是会订阅其他组件特定的 stream . 如果你想要订阅其他组件所有的 streams, 你必须一个个的订阅. [InputDeclarer](#) 有语法可以订阅默认 stream-id 的 stream, 代码: `declarer.shuffleGrouping ("1")`, 意思是: 订阅组件 “1” 的默认 stream, 等价于 `declarer.shuffleGrouping("1", DEFAULT_STREAM_ID)`.

Bolt 中最主要的方法是 `execute` 方法, 当有一个新 Tuple 输入的时候会进入这个方法. Bolt 使用[OutputCollector](#) 对象发送新的 Tuple. Bolt 必须在每一个 Tuple 处理完以后调用 [OutputCollector](#) 上的 `ack` 方法, Storm 就会知道 tuple 什么时候完成 (最终可以确定 调用源 Spout Tuple 是没有问题的). 当处理一个输入的 Tuple: 会基于这个 Tuple 产生零个或者多个 Tuple 发送出去, 当所有的tuple 完成后, 会调用 `acking`. Storm 提供了 [IBasicBolt](#) 接口会自动执行 `acking` .

最好在 Bolt 中启动新的线程异步处理 tuples. [OutputCollector](#) 是线程安全的, 并且可以在任何时刻调用.

相关资料:

- [IRichBolt](#): Bolts 的通用接口
- [IBasicBolt](#): 一个可以使用过滤或者一些简单功能的 Bolt 的接口
- [OutputCollector](#): Bolts 使用这个类的实例发送 Tuple 到他们的输出流
- [Guaranteeing message processing](#)

## Stream groupings

topology (拓扑) 定义中有一部分是为每一个 bolt 指定输入的 streams . stream grouping 定义了stream 如何在 Bolts tasks 之间分区.

Storm 中一共有8个内置的 Stream Grouping. 可以通过实现 [CustomStreamGrouping](#) 接口来自定义 Stream groupings.

1. **Shuffle grouping**: Tuple 随机的分发到 Bolt Task, 每个 Bolt 获取到等量的 Tuple.
2. **Fields grouping**: streams 通过 grouping 指定的字段来分区. 例如流通过 "user-id" 字段分区, 具有相同 "user-id" 的 Tuple 会发送到同一个task, 不同 "user-id" 的 Tuple 可能会流入到不同的 tasks.

3. **Partial Key grouping:** stream 通过 grouping 中指定的 field 来分组, 与 Fields Grouping 相似. 但是对于 2 个下游的 Bolt 来说是负载均衡的, 可以在输入数据不平均的情况下提供更好的优化. 以下地址 [This paper](#) 更好的解释了它是如何工作的及它的优势.
4. **All grouping:** stream 在所有的 Bolt Tasks 之间复制. 这个 Grouping 小心使用.
5. **Global grouping:** 整个 stream 会进入 Bolt 其中一个任务. 特别指出, 它会进入 id 最小的 task.
6. **None grouping:** 这个 grouping, 你不需要关心 stream 如何分组. 当前, None grouping 和 Shuffle grouping 等价. 同时, Storm 将使用 None grouping 的 bolts 和上游订阅的 bolt 和 spout 运行在同一个线程 (when possible).
7. **Direct grouping:** 这是一种特殊的 grouping 方式. stream 用这个方式 group 意味着由这个 Tuple 的生产者 来决定哪个 消费者 来接收它. Direct grouping 只能被用于 direct streams . 被发射到 direct stream 的 tuple 必须使用 `emitDirect(int, int, java.util.List)` 方法来发送. Bolt 可以使用 `TopologyContext` 或者通过保持对 `OutputCollector` (返回 Tuple 被发送到的目标 task id) 中的 `emit` 方法输出的跟踪, 获取到它的所有消费者的 ID .
8. **Local or shuffle grouping:** 如果目标 Bolt 有多个 task 和 streams 源 在同一个 worker 进程中, Tuple 只会 shuffle 到相同 worker 的任务. 否则, 就和 shuffle goruping 一样.

相关资料:

- [TopologyBuilder](#): 使用这个类来定义一个拓扑
- [InputDeclarer](#): 当在 `TopologyBuilder` 上调用 `setBolt` 方法的时候返回这个对象, 用于声明一个 Bolt 的 input streams 以及这些 streams 如何分组. **### 可靠性**

Storm 保障每一个 Spout 的 Tuple 都会被 topology (拓扑) 处理. 通过跟踪 tuples tree, 每个 spout tuple 都会触发 tree , 确保 tuples tree 成功完成. 每一个拓扑都有一个关联的“message timeout”. 如果 Storm 检测到一个 Spout Tuple 没有在这个超时时间内被处理完成, 则判定这个 Tuple 失败, 稍后重新执行.

要利用这个可靠性的功能, 当在 Tuple tree 中创建一个新的 edge , 必须告诉 Storm, 并且在一个单独的 tuple 完成时也要通知 Storm. 以上操作在 Bolt 用于发送 Tuple 的 `OutputCollector` 对象中完成这个操作. Anchoring (锚点) 在 `emit` 方法中完成, 使用 `ack` 方法来声明你已经成功完成了一个 Tuple 的处理.

更详细的说明, 请参阅 [保证消息容错](#).

## Tasks

每个 Spout 或者 Bolt 都以跨集群的多个 Task 方式执行. 每个 Task 对应一个 execution 的线程, stream groupings 定义如何从一个 Task 发送 Tuple 到另一个 Task. 可以在 TopologyBuilder 的 `setSpout` 和 `setBolt` 方法中为每个 Spout 或者 Bolt 设置并行度,.

## Workers

Topologies (拓扑) 在一个或者跨多个 worker 执行. 每个 Worker 进程是一个物理的 JVM, 执行 topology (拓扑) Tasks 中的一个子集. 例如, 如果一个拓扑的并行度是 300, 共有 50 个 Worker 在运行, 每个 Worker 会分配到 6 个 Task (作为 Worker 中的线程). Storm 会尽量把所有 Task 均匀的分配到所有的 Worker 上.

相关资料:

- [Config.TOPOLOGY\\_WORKERS](#): 这个配置项设置用于运行 topology (拓扑) 的 worker 数量.

# Scheduler (调度器)

Storm 现在有 4 种内置的 schedulers (调度器) : [DefaultScheduler](#), [IsolationScheduler](#), [MultitenantScheduler](#), [ResourceAwareScheduler](#).

## Pluggable scheduler (可插拔的调度器)

你可以实现你自己的 scheduler (调度器) 来替换掉默认的 scheduler (调度器) , 自定义分配executors 到 workers 的调度算法. 使用的时候, 在`storm.yaml` 文件中将 "`storm.scheduler`" 配置属性设置成你的class类, 并且您的 scheduler (调度器) 必须实现 [IScheduler](#) 接口。

## Isolation Scheduler (隔离调度器)

isolation scheduler (隔离调度器) 使得多个topologies (拓扑) 共享集群资源更加容易和安全. isolation scheduler (隔离调度器) 允许你指定某些 topologies (拓扑) 是 “isolated” (隔离的) , 这意味着这些 isolated topologies (隔离的拓扑) 运行在集群的特定机器上, 这些机器没有其他 topologies (拓扑) 运行。 isolated topologies (隔离的拓扑) 具有高优先级, 所以如果和 non-isolated topologies (非隔离的拓扑) 竞争资源, 资源将会分配给 isolated topologies (隔离的拓扑) , 如果必须给 isolated topologies (隔离的拓扑) 分配资源, 那么将会从 non-isolated topologies (非隔离的拓扑) 中抽取资源。一旦所有 isolated topologies (隔离的拓扑) 所需资源得到满足, 那么集群中剩下的机器将会被 non-isolated topologies (非隔离的拓扑) 共享。

您可以通过将 "`storm.scheduler`" 设置为 "`org.apache.storm.scheduler.IsolationScheduler`" , 这样 Nimbus 节点的 Scheduler 就配置为 Isolation Scheduler (隔离调度器) . 然后, 使用 "`isolation.scheduler.machines`" 配置来指定每个topology (拓扑) 分配多少台机器. 这个配置是从 topology name (拓扑名称) 到分配给此 topology (拓扑) 的隔离机器数量的 map 集合. 例如:

```
isolation.scheduler.machines:  
  "my-topology": 8  
  "tiny-topology": 1  
  "some-other-topology": 3
```

提交到集群中的topologies 如果没有出现在上述 map 集合中, 那么将不会被 isolated 。 请注意: user不可以设置 isolation 属性, 该配置只能通过集群的管理员分配 (这是故意这样设计的) 。

**isolation scheduler**（隔离调度器）通过在拓扑之间提供完全的隔离来解决多租户问题 - 避免 **topologies**（拓扑）之间的资源竞争问题. 最终的目的是 "**productionized**（生产黄精的）"  
**topologies**（拓扑）应该设置成 **isolated**，测试或开发中的 **topologies**（拓扑）不应该设置成 **isolated** 属性. 集群上的剩余机器可以为 **isolated topologies**（隔离的拓扑）提供故障切换，也可以用来运行 **non-isolated topologies**（非隔离的拓扑）.

# Configuration

Storm 具有各种各样的配置，用于调整 `nimbus`, `supervisors` 和运行 `topologies`（拓扑）的行为。一些配置是系统配置，不能在拓扑基础上对拓扑进行修改，而每个拓扑可以修改其他配置。

每个配置都有一个定义在 Storm 代码库 [defaults.yaml](#) 文件中的 `default value`（默认值）。您可以通过在 Nimbus 按 `supervisors` 的 `classpath` 中定义一个 `storm.yaml` 文件以覆盖那些配置。最后，您可以定义一个指定的 `topology` 配置，它在使用 [StormSubmitter](#) 时随着你的 `topology` 一起提交。然而，指定的 `topology` 配置只能够覆盖前缀 "TOPOLOGY" 开始的配置。

Storm 0.7.0 及以上版本允许您以 `per-bolt/per-spout` 为基础进行配置覆盖。可以通过这种方式覆盖的唯一配置是：

1. `"topology.debug"`
2. `"topology.max.spout.pending"`
3. `"topology.max.task.parallelism"`
4. `"topology.kryo.register"`: 这与其他工作有点不同，因为 `serializations`（序列化将）可用于拓扑中的所有组件。更多细节请参阅 [序列化](#)。

Java API 允许您以两种方式指定组件特定的配置：

1. *Internally*（内部的）：在任何 `spout` 或 `bolt` 中覆盖 `getComponentConfiguration` 方法，并返回组件指定的的配置 `map`。
2. *Externally*（外部的）：`TopSpringBuilder` 中的 `setSpout` 和 `setBolt` 返回一个可以用来覆盖组件配置的 `addConfiguration` 和 `addConfigurations` 方法的对象。

配置值的优先顺序是 `defaults.yaml` < `storm.yaml` < `topology` 指定的配置 < `internal`（内部）组件指定的配置 < `external`（外部）组件指定的配置。

## Resources:

- [Config](#): 所有配置的列表以及用于创建拓扑特定配置的辅助 `class`（类）
- [defaults.yaml](#): 所有配置的默认值
- [部署 Storm 集群](#): 介绍如何创建和配置 Storm 集群
- [在生产集上运行 topologies（拓扑）](#)：在集群上运行拓扑时列出了有用的配置
- [Local（本地）模式](#): 列出使用 `local model`（本地模式）时的有用配置

# Guaranteeing Message Processing

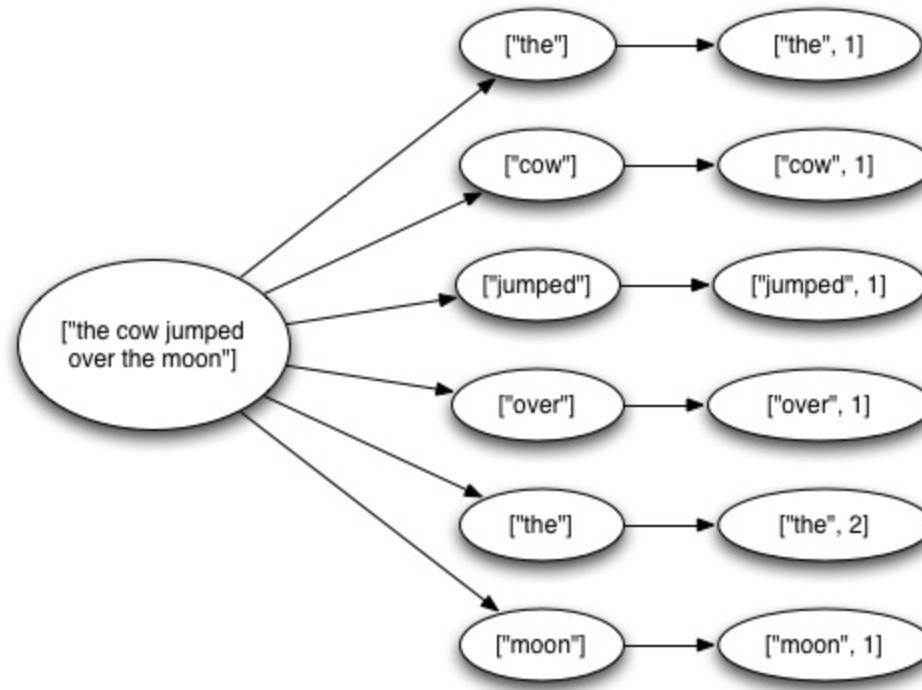
Storm 通过 Trident 对保证消息处理提供了不同的 level , 包括 best effort (尽力而为) , at least once (至少一次) 和exactly once (至少一次) . 这张页面描述如何保证至少处理一次.

**What does it mean for a message to be "fully processed"?** (一条信息被完全处理是什么意思)

一个 tuple 从 spout 流出, 可能会导致数以千计的 tuple 被创建.例如, 下面 streaming word count的 topology (拓扑) :

```
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("sentences", new KestrelSpout("kestrel.backtype.com",
                                              22133,
                                              "sentence_queue",
                                              new StringScheme()));
builder.setBolt("split", new SplitSentence(), 10)
    .shuffleGrouping("sentences");
builder.setBolt("count", new WordCount(), 20)
    .fieldsGrouping("split", new Fields("word"));
```

这个topology从 Kestrel queue 读取一行行的句子, 将句子按照空格分割成一个个的单词, 然后再发送之前计算的单词数量. 从 Spout 中流出的一个tuple 会触发创建许多 tuples: 一个 tuple 对应句子中的 word, 一个tuple对应每个 word 的 count。消息树像下面这样:



当 tuple tree 用完后且每个在 tree中的消息都被处理后, Storm 就认为从 spout 流出的 tuple

被完全处理了。

当一个 tuple tree 没有在特定的时间内完全处理，tuple就被认为是失败的。可以在指定的 topology (拓扑) 上使用 Config.TOPOLOGY\_MESSAGE\_TIMEOUT\_SECS 来配置这个超时时间，并且默认为30秒。

## What happens if a message is fully processed or fails to be fully processed? (如果消息完全处理或未能完全处理，会发生什么)

为了理解这个问题，我们来看一下 tuple 从 spout 流出后的声明周期。作为参考，这里是 spouts 实现的接口 (有关更多信息，请参阅 [Javadoc](#) ):

```
public interface ISpout extends Serializable {  
    void open(Map conf, TopologyContext context, SpoutOutputCollector collector);  
    void close();  
    void nextTuple();  
    void ack(Object msgId);  
    void fail(Object msgId);  
}
```

首先 Storm 请求一个 Spout 中的 tuple，使用 Spout 中的 nextTuple 方法。Spout 使用 SpoutOutputCollector 提供的 open 方法，发送 tuple 输出到 output streams 的其中一个。当发送 tuple 的时候，Spout 会提供一个 "message id"，用于以后标识 tuple。例如，KestrelSpout 从 kestrel queue 中读取消息，在发送消息的时候会提供一个 "message id"。发送一个 message 到 SpoutOutputCollector 像下面这样：

```
_collector.emit(new Values("field1", "field2", 3), msgId);
```

下一步，tuple 被发送到消费的 bolts 中，Storm 开始监控创建的 the tree of messages。如果 Storm 检测到 tuple 被完全处理，Storm 会在原来的 Spout 上根据 message id 调用 ack 方法。同样的，如果处理 tuple 超时了，Storm 会调用在 Spout 上调用 fail 方法。由于 tuple ack 或者 fail，都原来创建这个 tuple 的 Spout task 调用。所以如果一个 spout 在集群上运行很多任务，tuple 不会被多个 Spout 任务 acked 或者 failed。

我们再通过 KestrelSpout 来看看 Spout 需要保证消息处理那些情况。当 KestrelSpout 从 Kestrel queue 中读取消息的时候，它只是 "open" 了 message。这意味着消息并没有真正从队列出来，而是处于一种 "pending" 状态，承认 message 已经完成。在这种 pending 状态的时候，message 不会将消息发送到队列的其他用 consumer。此外，如果客户端断开所有 pending 状态的消息，那么客户端将把消息放回到队列。当一条 message opened 后，Kestrel 向客户端

提供消息，并提供一个唯一性的id标识消息。当发送 tuple 到 SpoutOutputCollector的时候，就用Kestrel 提供的id 作为“message id”。当 KestrelSpout 调用 ack或者fail的时候，KestrelSpout会向Kestrel 发送一条ack 或者 fail消息，包括 message id，以让Kestrel 将消息从队列中取出，或者放回去。

## What is Storm's reliability API?

用户想要保证Storm的可靠性，需要做两件事。第一，当你在 tuple tree 中创建新的 link 的时候，你需要告诉 Storm。第二，当你处理完一个独立的 tuple，也需要告诉Storm。做到这两件事情，Storm可以检测到 tree of tuples 什么时候完全处理了，并且可以适当的 ack 或者 fail spout tuples。Storm 的API提供了一个简单的方法来完成这两项任务。

在 tuple tree中指定一个link 的方法叫做 *anchoring*。当你发送一个新的 tuple 的时候就会执行 Anchoring 操作。我们使用下面的 bolt 作为一个例子。这个 bolt 将句子 tuple 分割成一个个 word tuple。

```
public class SplitSentence extends BaseRichBolt {  
    OutputCollector _collector;  
  
    public void prepare(Map conf, TopologyContext context, OutputCollector collector) {  
        _collector = collector;  
    }  
  
    public void execute(Tuple tuple) {  
        String sentence = tuple.getString(0);  
        for(String word: sentence.split(" ")) {  
            _collector.emit(tuple, new Values(word));  
        }  
        _collector.ack(tuple);  
    }  
  
    public void declareOutputFields(OutputFieldsDeclarer declarer) {  
        declarer.declare(new Fields("word"));  
    }  
}
```

通过指定输入 tuple 作为emit 的第一个参数，每一个word tuple 就会被 anchored。由于 word tuple 被 anchor，如果 word tuple 下游处理失败，tuple tree 的根节点会重新处理。相比之下，我们看一下 word tuple像下面这样发送。

```
_collector.emit(new Values(word));
```

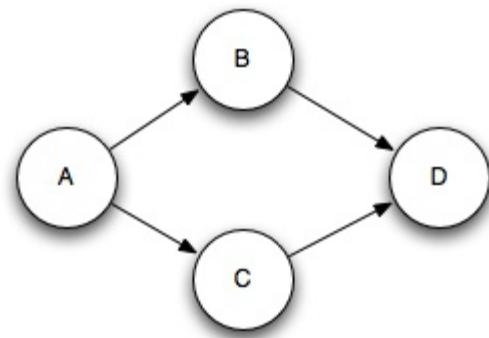
这种方式发送 word tuple 会导致 *unanchored*。如果tuple在处理下游的时候失败，根节点的 tuple不会重新处理。根据你的 topology（拓扑）需要来保证容错保证，有时候发送一个

unanchored tuple 也比较适合.

输出的tuple可以 anchor 多个input tuple, 当join和聚合的时候, 这是比较有用的一个 multi-anchored 的tuple处理失败后, 多个tuple都会被重新处理.通过指定一系列 tuples, 而不是单个tuple来完成 Multi-anchoring.例如:

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

Multi-anchoring 会添加 output tuple 到 multiple tuple trees.这就可能会破坏树形结构, 创建了tuple DAGs, 像这样:



Storm 的实现适用于DAG和树 (pre-release 只适用于trees,称为“tuple tree”)

Anchoring 是你如何指定 tuple tree的方式--在Storm可靠性 API 在 tuple tree中完成一个tuple后, 会执行下一部分的处理. 在OutputCollector上使用ack和fail方法的时候来完成.如果你回顾了SplitSentence例子, 你可以看到当所有的word tuple发送出去后, input tuple会被ack.

你可以使用 OutputCollector 的 fail 方法, 立即设置 tuple tree的根节点spout tuple为失败状态.例如, 你的应用可能数据库异常, 需要显式的 fail input tuple。通过显式的failing,spout tuple 可以比等待tuple超时速度更快.

你处理的每个tuple 必须acked或者failed.Storm使用内存监控每个tuple, 所以如果你必须fail 或者ack每个tuple,这个任务最终会out of memory.

许多bolts都会像下面这种模式读取 input tuple, 发送 tuples.在 execute 方法结束后acking tuple。bolts 有过滤或者一些简单的功能.Storm有一个接口叫做BasicBolt, 可以封装这些模式.SplitSentence示例可以写成BasicBolt, 如下所示:

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

```

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

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

```

该实现比以前的实现更简单，语义上相同。发送到 `BasicOutputCollector` 的 `tuple` 将自动 `anchor` 到输入元组，并且在执行方法完成时自动 `acked`。

相比之下，执行聚合或 `join` 的 `bolt` 可能会延迟 `ack input tuple`。聚合和 `join` 会多重 `anchor output tuples`。这些事情不在 `IBasicBolt` 的简单模式之上。

## How do I make my applications work correctly given that tuples can be replayed?

(如果重新处理 `tuples`，我如何使我的应用程序正常工作？)

与软件设计一样，答案是“`depend`”。如果你真的想要使用 `Trident` 保证 `exactly once` 语义。在某些情况下，像许多分析行为一样，丢弃数据是允许的，所以通过设置 `ackers bolts` 数量为 `0` `Config.TOPOLOGY_ACKERS`，来禁用容错。但在某些情况下，你想要确保所有的 `tuple` 至少处理一次，没有任何内容被丢弃。This is especially useful if all operations are idempotent or if deduping can happen afterwards. (废话可以不看)

## How does Storm implement reliability in an efficient way? (Storm 如何以有效的方式实现可靠性？)

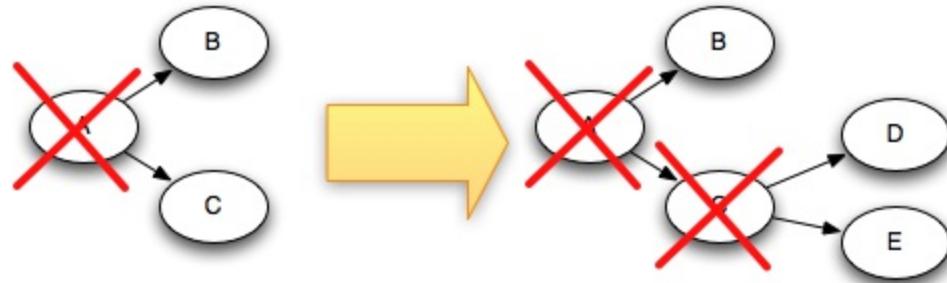
Storm 对于 Spout 的 `tuple` 都有一组特殊的 `acker` 任务用来跟踪 DAG tuples。当一个 `acker` 任务看到一个 DAG 完成后，就会发送一个消息到创建这个 spout tuple 的任务。你可以通过 `Config.TOPOLOGY_ACKERS` 来设置 `acker tasks` 的数量。Storm `TOPOLOGY_ACKERS` 配置默认是一个 worker 一个任务。

理解 Storm 可靠性实现最好的方式就是看 `tuple` 和 `tuple DAG` 的声明周期。当 topology 中创建了一个 `tuple`，也可以在一个 spout 或者 bolt，会给予一个 64 bit 的 id。这个 id 是 `ackers` 用来跟踪 `tuple DAG` 里的每个 spout tuple。

spout 中的每个 `tuple` 都知道他们 id，并存在于 `tuple tree` 中。当你发送一个新的 `tuple` 到 bolt 的时候，来自于 `tuple anchors`（锚点）的 spout tuple ids 会被复制到新的 `tuple`。当一个 `tuple`

被ack后，就会发送一条消息到 acker tasks，告知 tuple tree 应该如何改变。实际上，告诉 acker 的是“我已经完成了 tree 中的这个 spout tuple，这里是 tree 中 anchored（锚定）我的 tuples”。

例如，tuples D 和 E 是基于 tuple C 创建的，下面是当 tuple C 被 acked 后，tree 是如何改变的。



由于 tuple C 从 tree 中移除的同时，tuple "D" 和 "E" 都加入到 tree 中，所以 tree 永远不可能完成。

Storm 跟踪 tuple trees 有一些细节。如前所述，你可以在 topology 中定义任意数量的 acker 任务。这导致了以下问题：当一个元组在 topology 中被 acked 后，它如何知道是哪个 acker 任务发送的该消息？

Storm 使用 mod hash 的模式将 spout tuple id 映射到 acker task。由于每一个 tuple 都会带有 tree 中 id，所以他们知道和那个 acker task 进行通信。

Storm 的另一个细节就是如何让 Acker 任务跟踪他们正在跟踪的每个 spout tuple 的 spouts 任务。当一个 spout 任务发出一个新的元组时，它只需向相应的 acker 发送一条消息，告诉它它的任务 id 是负责这个新的元素的负责人。然后当 acker 看到树已经完成时，它知道要发送那个任务 id。

Acker tasks 不会显式的跟踪 tuple trees。对于具有成千上万个节点（或者更多）的大型 tuple tree，跟踪所有的 tuple trees 会占用很多内存。相反，ackers 采用不同的策略，只需要每个 spout tuple 固定空间（大约 20 字节）。这种跟踪算法是 Storm 工作的关键，也是重大突破之一。

一个 acker task 存储 spout tuple id 映射到一组值的 map，第一个值是创建这个 spout tuple 的任务 id。第二个值是称为“ack val”的 64 位数。ack val 表示整个 tuple tree 的状态，无论多大或

多小,它简化了在树中创建或者acked 的所有tuple ids 的xor.

当acker任务看到“ack val”已经变为0时，它知道元组树已经完成。由于 tuple ID是随机64位数，因此“ack val”突然变为0的机会非常小。从数学角度来看，每秒10K acks，就要花5万年，直到出错。即使如此，如果该 tuple 在拓扑中发生故障，则只会导致数据丢失。

现在，您了解可靠性算法，让我们回顾一下所有故障情况，并了解如何在每种情况下避免数据丢失：

- 一个tuple没有acked，因为对应的任务挂掉：在这种情况下，失败的元组会超时并重新处理。
- **Acker** 任务挂掉：这种情况下acker 跟踪的所有 spout tuples 都会超时并重新处理
- **Spout** 任务挂掉：在这种情况下，spout 的来源负责重新播放消息.例如，当客户端断开连接时，像Kestrel和RabbitMQ这样的队列会将所有挂起的消息放回队列。

正如你所看到的，Storm的可靠性机制是完全分布式，可扩展的和容错的。

## Tuning reliability（调整可靠性）

Acker任务是轻量级的，所以在 topology（拓扑）中您不需要非常多的task。您可以通过 Storm UI（组件ID“\_\_acker”）跟踪其性能。如果吞吐量看起来不正确，则需要添加更多的 acker任务。

如果可靠性对您不重要 -- 也就是说，您不关心在失败情况下丢失 tuple，那么您可以通过不跟踪 tuple tree来提高性能。没有跟踪tuple tree 将传递的消息数量减少一半，因为正常情况下，tuple tree 中的每个tuple都有一个确认消息。另外，它需要更少的id来保存在每个下游 tuple 中，从而减少带宽使用。

有三种方法来消除可靠性。第一个是将Config.TOPOLOGY\_ACKERS设置为0.在这种情况下，Storm会在 spout 发出一个tuple 后立即在 spout 上调用ack方法。元组树不会被跟踪。

第二种方法是通过消息消除消息的可靠性。您可以通过省略SpoutOutputCollector.emit方法中的消息标识来关闭单个 spout tuple 的跟踪。

最后，如果您不关心topology（拓扑）中下游的tuple的特定子集是否被处理，则可以将其作为无保留的 tuple发出。由于它们没有被锚定到任何 spout tuple，所以如果没有出现，它们不会导致任何 spout tuple失败。

# 守护进程容错

Storm 有几个不同的守护进程. 调度 workers 的 **Nimbus**, 启动和杀死 workers 的 **supervisors**, 可以访问日志的 **log viewer** (日志查看器) 以及显示集群状态 **UI**.

## 当一个 **worker** 挂掉时会发生什么?

当一个 worker 挂掉时, supervisor 将会重启它. 如果在启动它时继续发生故障并且没有发送 heartbeat (心跳) 给 Nimbus, 那么 Nimbus 将会重新调度 worker.

## 当一个 **node** (节点) 挂掉时会发生什么?

分配给该机器的 task (任务) 将超时, Nimbus 将这些 task (任务) 重新分配给其他机器.

## 当 **Nimbus** 或 **Supervisor** 挂掉时会发生什么?

Nimbus 和 Supervisor 守护进程是为 fail-fast (快速失败) (遇到任何意外情况时进程自毁) 和无状态 (所有状态都保存在 Zookeeper 或磁盘上) 而设计的. 像 [部署 Storm 集群](#) 中描述的一样, Nimbus 和 Supervisor 守护进程必须使用 daemontools 或 monit 工具进行监督. 所以如果 Nimbus 或 Supervisor 守护进程挂掉后, 它们会像什么都没发生一样重启.

最值得注意的是, 没有 worker 进程受到 Nimbus 或 Supervisors 挂掉的影响. 这与 Hadoop 相反, 如果 JobTracker 挂掉, 所有正在运行的 job 作业都将丢失.

## Nimbus 是单点故障的吗?

如果你失去了 Nimbus 节点, workers 仍然会继续工作. 此外, supervisors 如果挂掉, 将继续重新启动 workers. 但是, 如果没有 Nimbus, worker 在必要时不会重新分配给其他机器 (如失去 worker 机器).

Storm Nimbus 自 1.0.0 以来是 highly available (高可用的). 更多信息请参阅 [Nimbus HA 高可用设计](#) 文档.

## Storm 是如何保证数据处理的?

Storm 提供了保证数据处理的机制, 即使节点挂掉或 messages (消息) 丢失. 更多细节请参阅 [保证消息处理](#).

# 命令行客户端

---

此页面描述了 "Storm" 命令行客户端可能使用的所有命令. 要了解如何设置 "风暴" 客户端与远程群集的操作, 请按照 [安装开发环境](#) 中的说明进行操作.

这些命令是:

1. jar
2. sql
3. kill
4. activate
5. deactivate
6. rebalance
7. repl
8. classpath
9. localconfvalue
0. remoteconfvalue
1. nimbus
2. supervisor
3. ui
4. drpc
5. blobstore
6. dev-zookeeper
7. get-errors
8. heartbeats
9. kill\_workers
0. list
1. logviewer
2. monitor
3. node-health-check
4. pacemaker
5. set\_log\_level
6. shell

7. upload-credentials
8. version
9. help

## jar

Syntax: `storm jar topology-jar-path class ...`

Runs the main method of `class` with the specified arguments. The storm jars and configs in `~/.storm` are put on the classpath. The process is configured so that [StormSubmitter](#) will upload the jar at `topology-jar-path` when the topology is submitted.

When you want to ship other jars which is not included to application jar, you can pass them to `--jars` option with comma-separated string. For example, `--jars "your-local-jar.jar,your-local-jar2.jar"` will load your-local-jar.jar and your-local-jar2.jar. And when you want to ship maven artifacts and its transitive dependencies, you can pass them to `--artifacts` with comma-separated string. You can also exclude some dependencies like what you're doing in maven pom. Please add exclusion artifacts with '^' separated string after the artifact. For example,

```
--artifacts "redis.clients:jedis:2.9.0,org.apache.kafka:kafka_2.10:0.8.2.2^org.slf4j:slf4j-log4j12"
```

will load jedis and kafka artifact and all of transitive dependencies but exclude slf4j-log4j12 from kafka.

When you need to pull the artifacts from other than Maven Central, you can pass remote repositories to `--artifactRepositories` option with comma-separated string. Repository format is "^". '^' is taken as separator because URL allows various characters. For example, `--artifactRepositories "jboss-repository^http://repository.jboss.com/maven2,HDPRepo^http://repo.hortonworks.com/content"` will add JBoss and HDP repositories for dependency resolver.

Complete example of both options is here:

```
./bin/storm jar example/storm-starter/storm-starter-topologies-*.jar org.apache.storm.starter.Rollin
```

When you pass jars and/or artifacts options, StormSubmitter will upload them when the topology is submitted, and they will be included to classpath of both the process which runs the class, and also workers for that topology.

## sql

Syntax: `storm sql sql-file topology-name`

Compiles the SQL statements into a Trident topology and submits it to Storm.

`--jars` and `--artifacts`, and `--artifactRepositories` options available for jar are also applied to sql command. Please refer "help jar" to see how to use `--jars` and `--artifacts`, and `--artifactRepositories` options. You normally want to pass these options since you need to set data source to your sql which is an external storage in many cases.

## kill

Syntax: `storm kill topology-name [-w wait-time-secs]`

Kills the topology with the name `topology-name`. Storm will first deactivate the topology's spouts for the duration of the topology's message timeout to allow all messages currently being processed to finish processing. Storm will then shutdown the workers and clean up their state. You can override the length of time Storm waits between deactivation and shutdown with the `-w` flag.

## activate

Syntax: `storm activate topology-name`

Activates the specified topology's spouts.

## deactivate

Syntax: `storm deactivate topology-name`

Deactivates the specified topology's spouts.

## rebalance

Syntax:

```
storm rebalance topology-name [-w wait-time-secs] [-n new-num-workers] [-e component=parallelism]*
```

Sometimes you may wish to spread out where the workers for a topology are running. For example, let's say you have a 10 node cluster running 4 workers per node, and then let's say you add another 10 nodes to the cluster. You may wish to have Storm spread out the

workers for the running topology so that each node runs 2 workers. One way to do this is to kill the topology and resubmit it, but Storm provides a "rebalance" command that provides an easier way to do this.

Rebalance will first deactivate the topology for the duration of the message timeout (overridable with the -w flag) and then redistribute the workers evenly around the cluster. The topology will then return to its previous state of activation (so a deactivated topology will still be deactivated and an activated topology will go back to being activated).

The rebalance command can also be used to change the parallelism of a running topology. Use the -n and -e switches to change the number of workers or number of executors of a component respectively.

## repl

Syntax: `storm repl`

Opens up a Clojure REPL with the storm jars and configuration on the classpath. Useful for debugging.

## classpath

Syntax: `storm classpath`

Prints the classpath used by the storm client when running commands.

## localconfvalue

Syntax: `storm localconfvalue conf-name`

Prints out the value for `conf-name` in the local Storm configs. The local Storm configs are the ones in `~/.storm/storm.yaml` merged in with the configs in `defaults.yaml`.

## remoteconfvalue

Syntax: `storm remoteconfvalue conf-name`

Prints out the value for `conf-name` in the cluster's Storm configs. The cluster's Storm configs are the ones in `$STORM-PATH/conf/storm.yaml` merged in with the configs in `defaults.yaml`. This command must be run on a cluster machine.

## nimbus

Syntax: `storm nimbus`

Launches the nimbus daemon. This command should be run under supervision with a tool like [daemontools](#) or [monit](#). See [Setting up a Storm cluster](#) for more information.

## supervisor

Syntax: `storm supervisor`

Launches the supervisor daemon. This command should be run under supervision with a tool like [daemontools](#) or [monit](#). See [Setting up a Storm cluster](#) for more information.

## ui

Syntax: `storm ui`

Launches the UI daemon. The UI provides a web interface for a Storm cluster and shows detailed stats about running topologies. This command should be run under supervision with a tool like [daemontools](#) or [monit](#). See [Setting up a Storm cluster](#) for more information.

## drpc

Syntax: `storm drpc`

Launches a DRPC daemon. This command should be run under supervision with a tool like [daemontools](#) or [monit](#). See [Distributed RPC](#) for more information.

## blobstore

Syntax: `storm blobstore cmd`

list [KEY...] - lists blobs currently in the blob store

cat [-f FILE] KEY - read a blob and then either write it to a file, or STDOUT (requires read access).

create [-f FILE] [-a ACL ...] [--replication-factor NUMBER] KEY - create a new blob.

Contents comes from a FILE or STDIN. ACL is in the form [uo]:[username]:[r-][w-][a-] can be comma separated list.

update [-f FILE] KEY - update the contents of a blob. Contents comes from a FILE or STDIN (requires write access).

delete KEY - delete an entry from the blob store (requires write access).

set-acl [-s ACL] KEY - ACL is in the form [uo]:[username]:[r-][w-][a-] can be comma separated list (requires admin access).

replication --read KEY - Used to read the replication factor of the blob.

replication --update --replication-factor NUMBER KEY where NUMBER > 0. It is used to update the replication factor of a blob.

For example, the following would create a mytopo:data.tgz key using the data stored in data.tgz. User alice would have full access, bob would have read/write access and everyone else would have read access.

```
storm blobstore create mytopo:data.tgz -f data.tgz -a u:alice:rwa,u:bob:rw,o::r
```

See [Blobstore\(Distcahce\)](#) for more information.

## dev-zookeeper

Syntax: `storm dev-zookeeper`

Launches a fresh Zookeeper server using "dev.zookeeper.path" as its local dir and "storm.zookeeper.port" as its port. This is only intended for development/testing, the Zookeeper instance launched is not configured to be used in production.

## get-errors

Syntax: `storm get-errors topology-name`

Get the latest error from the running topology. The returned result contains the key value pairs for component-name and component-error for the components in error. The result is returned in json format.

## heartbeats

Syntax: `storm heartbeats [cmd]`

**list PATH** - lists heartbeats nodes under PATH currently in the ClusterState. **get PATH** - Get the heartbeat data at PATH

## **kill\_workers**

Syntax: `storm kill_workers`

Kill the workers running on this supervisor. This command should be run on a supervisor node. If the cluster is running in secure mode, then user needs to have admin rights on the node to be able to successfully kill all workers.

## **list**

Syntax: `storm list`

List the running topologies and their statuses.

## **logviewer**

Syntax: `storm logviewer`

Launches the log viewer daemon. It provides a web interface for viewing storm log files. This command should be run under supervision with a tool like daemontools or monit.

See Setting up a Storm cluster for more information.

(<http://storm.apache.org/documentation/Setting-up-a-Storm-cluster>)

## **monitor**

Syntax:

```
storm monitor topology-name [-i interval-secs] [-m component-id] [-s stream-id] [-w [emitted | transferred]]
```

Monitor given topology's throughput interactively. One can specify poll-interval, component-id, stream-id, watch-item[emitted | transferred] By default, poll-interval is 4 seconds; all component-ids will be list; stream-id is 'default'; watch-item is 'emitted';

## **node-health-check**

Syntax: `storm node-health-check`

Run health checks on the local supervisor.

## **pacemaker**

Syntax: `storm pacemaker`

Launches the Pacemaker daemon. This command should be run under supervision with a tool like daemontools or monit.

See Setting up a Storm cluster for more information.

(<http://storm.apache.org/documentation/Setting-up-a-Storm-cluster>)

## **set\_log\_level**

Syntax:

```
storm set_log_level -l [logger name]=[log level][:optional timeout] -r [logger name] topology-name
```

Dynamically change topology log levels

where log level is one of: ALL, TRACE, DEBUG, INFO, WARN, ERROR, FATAL, OFF and timeout is integer seconds.

e.g. `./bin/storm set_log_level -l ROOT=DEBUG:30 topology-name`

Set the root logger's level to DEBUG for 30 seconds

`./bin/storm set_log_level -l com.myapp=WARN topology-name`

Set the com.myapp logger's level to WARN for 30 seconds

`./bin/storm set_log_level -l com.myapp=WARN -l com.myOtherLogger=ERROR:123 topology-name`

Set the com.myapp logger's level to WARN indefinitely, and com.myOtherLogger to ERROR for 123 seconds

`./bin/storm set_log_level -r com.myOtherLogger topology-name`

Clears settings, resetting back to the original level

## **shell**

Syntax: `storm shell resourcesdir command args`

Makes constructing jar and uploading to nimbus for using non JVM languages

eg: `storm shell resources/ python topology.py arg1 arg2`

## **upload-credentials**

Syntax: `storm upload_credentials topology-name [credkey credvalue]*`

Uploads a new set of credentials to a running topology

## **version**

Syntax: `storm version`

Prints the version number of this Storm release.

## **help**

Syntax: `storm help [command]`

Print one help message or list of available commands

# Storm UI REST API

Storm UI 守护程序提供了一个 REST API, 允许您与 Storm 集群进行交互, 其中包括检索 metrics (度量) 数据和配置信息, 启动或停止 topologies (拓扑) 的管理操作。

## 数据格式

该 REST API 返回 JSON 响应并支持 JSONP. 客户端可以传回一个回调查询参数, 以在回调函数中包装 JSON。 **REST API allows CORS by default.**

## 使用 UI REST API

**Note** (注意) : 建议忽略 JSON 响应中的 *undocumented elements* (未记录元素), 因为未来版本的 *Storm* 可能不再 支持这些元素.

## REST API Base URL

REST API 是 Storm 的 UI 守护进程 (由 "storm ui" 启动) 的一部分, 因此与 Storm UI 在同一主机和端口上运行 (UI 守护程序通常与 Nimbus 守护程序在同一主机上运行). 通过 `ui.port` 来配置 port (端口), 它默认设置为 8080 (请参阅 [defaults.yaml](#)) .

该 API 的 base URL 将是:

```
http://<ui-host>:<ui-port>/api/v1/...
```

您可以使用 `curl` 这样的工具来操作 REST API:

```
# 请求集群配置
# Note (注意) : 我们假设 ui.port 配置的默认值是 8080.
$ curl http://<ui-host>:8080/api/v1/cluster/configuration
```

## 在安全的环境中模仿用户

在安全环境中, 经过身份验证的用户可以模拟另一个用户. 为了模拟用户, 调用者必须通过 `doAsUser` 参数或 `header`, 其值设置为用户需要执行该 `request` 请求. 请看 [SECURITY.MD](#) 以了解有关如何设置模拟 ACL 和授权的更多信息. 其余的 API 使用与 nimbus 使用的相同的配置和 acls.

示例:

1. `http://ui-daemon-host-name:8080/api/v1/topology/wordcount-1-1425844354\?doAsUser=testUser1`
2. `curl 'http://localhost:8080/api/v1/topology/wordcount-1-1425844354/activate' -X POST -H 'doAsUs`

# GET Operations

## /api/v1/cluster/configuration (GET)

Returns the cluster configuration.

Sample response (does not include all the data fields):

```
{  
    "dev.zookeeper.path": "/tmp/dev-storm-zookeeper",  
    "topology.tick.tuple.freq.secs": null,  
    "topology.builtin.metrics.bucket.size.secs": 60,  
    "topology.fall.back.on.java.serialization": true,  
    "topology.max.error.report.per.interval": 5,  
    "zmq.linger.millis": 5000,  
    "topology.skip.missing.kryo.registrations": false,  
    "storm.messaging.netty.client_worker_threads": 1,  
    "ui.childopts": "-Xmx768m",  
    "storm.zookeeper.session.timeout": 20000,  
    "nimbus.reassign": true,  
    "topology.trident.batch.emit.interval.millis": 500,  
    "storm.messaging.netty.flush.check.interval.ms": 10,  
    "nimbus.monitor.freq.secs": 10,  
    "logviewer.childopts": "-Xmx128m",  
    "java.library.path": "/usr/local/lib:/opt/local/lib:/usr/lib",  
    "topology.executor.send.buffer.size": 1024,  
}
```

## /api/v1/cluster/summary (GET)

Returns cluster summary information such as nimbus uptime or number of supervisors.

Response fields:

Field	Value	Description
stormVersion	String	Storm version
supervisors	Integer	Number of supervisors running
topologies	Integer	Number of topologies running
slotsTotal	Integer	Total number of available worker slots
slotsUsed	Integer	Number of worker slots used
slotsFree	Integer	Number of worker slots available
executorsTotal	Integer	Total number of executors

tasksTotal	Integer	Total tasks
schedulerDisplayResource	Boolean	Whether to display scheduler resource information
totalMem	Double	The total amount of memory in the cluster in MB
totalCpu	Double	The total amount of CPU in the cluster
availMem	Double	The amount of available memory in the cluster in MB
availCpu	Double	The amount of available cpu in the cluster
memAssignedPercentUtil	Double	The percent utilization of assigned memory resources in cluster
cpuAssignedPercentUtil	Double	The percent utilization of assigned CPU resources in cluster

Sample response:

```
{
  "stormVersion": "0.9.2-incubating-SNAPSHOT",
  "supervisors": 1,
  "slotsTotal": 4,
  "slotsUsed": 3,
  "slotsFree": 1,
  "executorsTotal": 28,
  "tasksTotal": 28,
  "schedulerDisplayResource": true,
  "totalMem": 4096.0,
  "totalCpu": 400.0,
  "availMem": 1024.0,
  "availCPU": 250.0,
  "memAssignedPercentUtil": 75.0,
  "cpuAssignedPercentUtil": 37.5
}
```

## /api/v1/supervisor/summary (GET)

Returns summary information for all supervisors.

Response fields:

Field	Value	Description
id	String	Supervisor's id

host	String	Supervisor's host name
uptime	String	Shows how long the supervisor is running
uptimeSeconds	Integer	Shows how long the supervisor is running in seconds
slotsTotal	Integer	Total number of available worker slots for this supervisor
slotsUsed	Integer	Number of worker slots used on this supervisor
schedulerDisplayResource	Boolean	Whether to display scheduler resource information
totalMem	Double	Total memory capacity on this supervisor
totalCpu	Double	Total CPU capacity on this supervisor
usedMem	Double	Used memory capacity on this supervisor
usedCpu	Double	Used CPU capacity on this supervisor

Sample response:

```
{
  "supervisors": [
    {
      "id": "0b879808-2a26-442b-8f7d-23101e0c3696",
      "host": "10.11.1.7",
      "uptime": "5m 58s",
      "uptimeSeconds": 358,
      "slotsTotal": 4,
      "slotsUsed": 3,
      "totalMem": 3000,
      "totalCpu": 400,
      "usedMem": 1280,
      "usedCPU": 160
    }
  ],
  "schedulerDisplayResource": true
}
```

## /api/v1/nimbus/summary (GET)

Returns summary information for all nimbus hosts.

Response fields:

Field	Value	Description
host	String	Nimbus' host name
port	int	Nimbus' port number
status	String	Possible values are Leader, Not a Leader, Dead
nimbusUpTime	String	Shows since how long the nimbus has been running
nimbusUpTimeSeconds	String	Shows since how long the nimbus has been running in seconds
nimbusLogLink	String	Logviewer url to view the nimbus.log
version	String	Version of storm this nimbus host is running

Sample response:

```
{
  "nimbususes": [
    {
      "host": "192.168.202.1",
      "port": 6627,
      "nimbusLogLink": "http://192.168.202.1:8000/log?file=nimbus.log",
      "status": "Leader",
      "version": "0.10.0-SNAPSHOT",
      "nimbusUpTime": "3m 33s",
      "nimbusUpTimeSeconds": "213"
    }
  ]
}
```

## /api/v1/history/summary (GET)

Returns a list of all running topologies' IDs submitted by the current user.

Response fields:

Field	Value	Description
topo-history	List	List of Topologies' IDs

Sample response:

```
{
  "topo-history": [
    "wc6-1-1446571009",
    "wc8-2-1446587178"
  ]
}
```

```
    ]  
}
```

## /api/v1/supervisor (GET)

Returns summary for a supervisor by id, or all supervisors running on a host.

Examples:

- ```
1. By host: http://ui-daemon-host-name:8080/api/v1/supervisor?host=supervisor-daemon-host-name  
2. By id: http://ui-daemon-host-name:8080/api/v1/supervisor?id=f5449110-1daa-43e2-89e3-69917b16dec
```

Request parameters:

| Parameter | Value                                  | Description                                                                                                                       |
|-----------|----------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------|
| id        | String.<br>Supervisor id               | If specified, respond with the supervisor and worker stats with id. Note that when id is specified, the host argument is ignored. |
| host      | String. Host name                      | If specified, respond with all supervisors and worker stats in the host (normally just one)                                       |
| sys       | String. Values 1 or 0. Default value 0 | Controls including sys stats part of the response                                                                                 |

Response fields:

| Field                    | Value   | Description                                       |
|--------------------------|---------|---------------------------------------------------|
| supervisors              | Array   | Array of supervisor summaries                     |
| workers                  | Array   | Array of worker summaries                         |
| schedulerDisplayResource | Boolean | Whether to display scheduler resource information |

Each supervisor is defined by:

| Field | Value  | Description            |
|-------|--------|------------------------|
| id    | String | Supervisor's id        |
| host  | String | Supervisor's host name |

|                          |         |                                                     |
|--------------------------|---------|-----------------------------------------------------|
| uptime                   | String  | Shows how long the supervisor is running            |
| uptimeSeconds            | Integer | Shows how long the supervisor is running in seconds |
| slotsTotal               | Integer | Total number of worker slots for this supervisor    |
| slotsUsed                | Integer | Number of worker slots used on this supervisor      |
| schedulerDisplayResource | Boolean | Whether to display scheduler resource information   |
| totalMem                 | Double  | Total memory capacity on this supervisor            |
| totalCpu                 | Double  | Total CPU capacity on this supervisor               |
| usedMem                  | Double  | Used memory capacity on this supervisor             |
| usedCpu                  | Double  | Used CPU capacity on this supervisor                |

Each worker is defined by:

| Field              | Value      | Description                                             |
|--------------------|------------|---------------------------------------------------------|
| supervisorId       | String     | Supervisor's id                                         |
| host               | String     | Worker's host name                                      |
| port               | Integer    | Worker's port                                           |
| topologyId         | String     | Topology Id                                             |
| topologyName       | String     | Topology Name                                           |
| executorsTotal     | Integer    | Number of executors used by the topology in this worker |
| assignedMemOnHeap  | Double     | Assigned On-Heap Memory by Scheduler (MB)               |
| assignedMemOffHeap | Double     | Assigned Off-Heap Memory by Scheduler (MB)              |
| assignedCpu        | Number     | Assigned CPU by Scheduler (%)                           |
| componentNumTasks  | Dictionary | Components -> # of executing tasks                      |
| uptime             | String     | Shows how long the worker is running                    |

|               |         |                                                 |
|---------------|---------|-------------------------------------------------|
| uptimeSeconds | Integer | Shows how long the worker is running in seconds |
| workerLogLink | String  | Link to worker log viewer page                  |

Sample response:

```
{
  "supervisors": [
    {
      "totalMem": 4096.0,
      "host": "192.168.10.237",
      "id": "bdfe8eff-f1d8-4bce-81f5-9d3ae1bf432e-169.254.129.212",
      "uptime": "7m 8s",
      "totalCpu": 400.0,
      "usedCpu": 495.0,
      "usedMem": 3432.0,
      "slotsUsed": 2,
      "version": "0.10.1",
      "slotsTotal": 4,
      "uptimeSeconds": 428
    }],
    "schedulerDisplayResource": true,
    "workers": [
      {
        "topologyName": "ras",
        "topologyId": "ras-4-1460229987",
        "host": "192.168.10.237",
        "supervisorId": "bdfe8eff-f1d8-4bce-81f5-9d3ae1bf432e-169.254.129.212",
        "assignedMemOnHeap": 704.0,
        "uptime": "2m 47s",
        "uptimeSeconds": 167,
        "port": 6707,
        "workerLogLink": "http://192.168.10.237:8000/log?file=ras-4-1460229987%2F6707%2Fworker.log",
        "componentNumTasks": {
          "word": 5
        },
        "executorsTotal": 8,
        "assignedCpu": 130.0,
        "assignedMemOffHeap": 80.0
      },
      {
        "topologyName": "ras",
        "topologyId": "ras-4-1460229987",
        "host": "192.168.10.237",
        "supervisorId": "bdfe8eff-f1d8-4bce-81f5-9d3ae1bf432e-169.254.129.212",
        "assignedMemOnHeap": 904.0,
        "uptime": "2m 53s",
        "port": 6706,
        "workerLogLink": "http://192.168.10.237:8000/log?file=ras-4-1460229987%2F6706%2Fworker.log",
        "componentNumTasks": {
          "exclaim2": 2,
          "exclaim1": 3,
          "word": 5
        },
        "executorsTotal": 10,
        "assignedMemOffHeap": 80.0
      }
    ]
}
```

```

    "uptimeSeconds":173,
    "assignedCpu":165.0,
    "assignedMemOffHeap":80.0
  }]
}

```

## /api/v1/topology/summary (GET)

Returns summary information for all topologies.

Response fields:

| Field               | Value   | Description                                                      |
|---------------------|---------|------------------------------------------------------------------|
| id                  | String  | Topology Id                                                      |
| name                | String  | Topology Name                                                    |
| status              | String  | Topology Status                                                  |
| uptime              | String  | Shows how long the topology is running                           |
| uptimeSeconds       | Integer | Shows how long the topology is running in seconds                |
| tasksTotal          | Integer | Total number of tasks for this topology                          |
| workersTotal        | Integer | Number of workers used for this topology                         |
| executorsTotal      | Integer | Number of executors used for this topology                       |
| replicationCount    | Integer | Number of nimbus hosts on which this topology code is replicated |
| requestedMemOnHeap  | Double  | Requested On-Heap Memory by User (MB)                            |
| requestedMemOffHeap | Double  | Requested Off-Heap Memory by User (MB)                           |
| requestedTotalMem   | Double  | Requested Total Memory by User (MB)                              |
| requestedCpu        | Double  | Requested CPU by User (%)                                        |
| assignedMemOnHeap   | Double  | Assigned On-Heap Memory by Scheduler (MB)                        |
| assignedMemOffHeap  | Double  | Assigned Off-Heap Memory by Scheduler (MB)                       |
| assignedTotalMem    | Double  | Assigned Total Memory by Scheduler (MB)                          |

|                          |         |                                                   |
|--------------------------|---------|---------------------------------------------------|
| assignedCpu              | Double  | Assigned CPU by Scheduler (%)                     |
| schedulerDisplayResource | Boolean | Whether to display scheduler resource information |

Sample response:

```
{
  "topologies": [
    {
      "id": "WordCount3-1-1402960825",
      "name": "WordCount3",
      "status": "ACTIVE",
      "uptime": "6m 5s",
      "uptimeSeconds": 365,
      "tasksTotal": 28,
      "workersTotal": 3,
      "executorsTotal": 28,
      "replicationCount": 1,
      "requestedMemOnHeap": 640,
      "requestedMemOffHeap": 128,
      "requestedTotalMem": 768,
      "requestedCpu": 80,
      "assignedMemOnHeap": 640,
      "assignedMemOffHeap": 128,
      "assignedTotalMem": 768,
      "assignedCpu": 80
    }
  ],
  "schedulerDisplayResource": true
}
```

## /api/v1/topology-workers/:id (GET)

Returns the worker' information (host and port) for a topology.

Response fields:

| Field        | Value   | Description                         |
|--------------|---------|-------------------------------------|
| hostPortList | List    | Workers' information for a topology |
| name         | Integer | Logviewer Port                      |

Sample response:

```
{
  "hostPortList": [
    {
      "host": "192.168.202.2",
      "port": 4150
    }
  ]
}
```

```

        "port":6701
    },
    {
        "host":"192.168.202.2",
        "port":6702
    },
    {
        "host":"192.168.202.3",
        "port":6700
    }
],
"logviewerPort":8000
}

```

## /api/v1/topology/:id (GET)

Returns topology information and statistics. Substitute id with topology id.

Request parameters:

| Parameter | Value                                  | Description                                       |
|-----------|----------------------------------------|---------------------------------------------------|
| id        | String (required)                      | Topology Id                                       |
| window    | String. Default value :all-time        | Window duration for metrics in seconds            |
| sys       | String. Values 1 or 0. Default value 0 | Controls including sys stats part of the response |

Response fields:

| Field         | Value   | Description                                       |
|---------------|---------|---------------------------------------------------|
| id            | String  | Topology Id                                       |
| name          | String  | Topology Name                                     |
| uptime        | String  | How long the topology has been running            |
| uptimeSeconds | Integer | How long the topology has been running in seconds |
| status        | String  | Current status of the topology, e.g. "ACTIVE"     |
| tasksTotal    | Integer | Total number of tasks for this topology           |

|                          |         |                                                                                |
|--------------------------|---------|--------------------------------------------------------------------------------|
| workersTotal             | Integer | Number of workers used for this topology                                       |
| executorsTotal           | Integer | Number of executors used for this topology                                     |
| msgTimeout               | Integer | Number of seconds a tuple has before the spout considers it failed             |
| windowHint               | String  | window param value in "hh mm ss" format. Default value is "All Time"           |
| schedulerDisplayResource | Boolean | Whether to display scheduler resource information                              |
| replicationCount         | Integer | Number of nimbus hosts on which this topology code is replicated               |
| debug                    | Boolean | If debug is enabled for the topology                                           |
| samplingPct              | Double  | Controls downsampling of events before they are sent to event log (percentage) |
| assignedMemOnHeap        | Double  | Assigned On-Heap Memory by Scheduler (MB)                                      |
| assignedMemOffHeap       | Double  | Assigned Off-Heap Memory by Scheduler (MB)                                     |
| assignedTotalMem         | Double  | Assigned Off-Heap + On-Heap Memory by Scheduler(MB)                            |
| assignedCpu              | Double  | Assigned CPU by Scheduler(%)                                                   |
| requestedMemOnHeap       | Double  | Requested On-Heap Memory by User (MB)                                          |
| requestedMemOffHeap      | Double  | Requested Off-Heap Memory by User (MB)                                         |
| requestedCpu             | Double  | Requested CPU by User (%)                                                      |
| topologyStats            | Array   | Array of all the topology related stats per time window                        |

|                               |                                                 |                                                         |
|-------------------------------|-------------------------------------------------|---------------------------------------------------------|
| topologyStats.windowPretty    | String                                          | Duration passed in HH:MM:SS format                      |
| topologyStats.window          | String                                          | User requested time window for metrics                  |
| topologyStats.emitted         | Long                                            | Number of messages emitted in given window              |
| topologyStats.transferred     | Long                                            | Number messages transferred in given window             |
| topologyStats.completeLatency | String (double value returned in String format) | Total latency for processing the message                |
| topologyStats.acked           | Long                                            | Number of messages acked in given window                |
| topologyStats.failed          | Long                                            | Number of messages failed in given window               |
| workers                       | Array                                           | Array of workers in topology                            |
| workers.supervisorId          | String                                          | Supervisor's id                                         |
| workers.host                  | String                                          | Worker's host name                                      |
| workers.port                  | Integer                                         | Worker's port                                           |
| workers.topologyId            | String                                          | Topology Id                                             |
| workers.topologyName          | String                                          | Topology Name                                           |
| workers.executorsTotal        | Integer                                         | Number of executors used by the topology in this worker |
| workers.assignedMemOnHeap     | Double                                          | Assigned On-Heap Memory by Scheduler (MB)               |
| workers.assignedMemOffHeap    | Double                                          | Assigned Off-Heap Memory by Scheduler (MB)              |
| workers.assignedCpu           | Number                                          | Assigned CPU by Scheduler (%)                           |
| workers.componentNumTasks     | Dictionary                                      | Components -> # of executing tasks                      |

|                           |                                                 |                                                                     |
|---------------------------|-------------------------------------------------|---------------------------------------------------------------------|
| workers.uptime            | String                                          | Shows how long the worker is running                                |
| workers.uptimeSeconds     | Integer                                         | Shows how long the worker is running in seconds                     |
| workers.workerLogLink     | String                                          | Link to worker log viewer page                                      |
| spouts                    | Array                                           | Array of all the spout components in the topology                   |
| spouts.spoutId            | String                                          | Spout id                                                            |
| spouts.executors          | Integer                                         | Number of executors for the spout                                   |
| spouts.emitted            | Long                                            | Number of messages emitted in given window                          |
| spouts.completeLatency    | String (double value returned in String format) | Total latency for processing the message                            |
| spouts.transferred        | Long                                            | Total number of messages transferred in given window                |
| spouts.tasks              | Integer                                         | Total number of tasks for the spout                                 |
| spouts.lastError          | String                                          | Shows the last error happened in a spout                            |
| spouts.errorLapsedSecs    | Integer                                         | Number of seconds elapsed since that last error happened in a spout |
| spouts.errorWorkerLogLink | String                                          | Link to the worker log that reported the exception                  |
| spouts.acked              | Long                                            | Number of messages acked                                            |
| spouts.failed             | Long                                            | Number of messages failed                                           |
| spouts.requestedMemOnHeap | Double                                          | Requested On-Heap Memory by User (MB)                               |
|                           |                                                 | Requested Off-Heap Memory by                                        |

|                            |                                                 |                                                                                          |
|----------------------------|-------------------------------------------------|------------------------------------------------------------------------------------------|
| spouts.requestedMemOffHeap | Double                                          | User (MB)                                                                                |
| spouts.requestedCpu        | Double                                          | Requested CPU by User (%)                                                                |
| bolts                      | Array                                           | Array of bolt components in the topology                                                 |
| bolts.boltId               | String                                          | Bolt id                                                                                  |
| bolts.capacity             | String (double value returned in String format) | This value indicates number of messages executed * average execute latency / time window |
| bolts.processLatency       | String (double value returned in String format) | Average time of the bolt to ack a message after it was received                          |
| bolts.executeLatency       | String (double value returned in String format) | Average time to run the execute method of the bolt                                       |
| bolts.executors            | Integer                                         | Number of executor tasks in the bolt component                                           |
| bolts.tasks                | Integer                                         | Number of instances of bolt                                                              |
| bolts.acked                | Long                                            | Number of tuples acked by the bolt                                                       |
| bolts.failed               | Long                                            | Number of tuples failed by the bolt                                                      |
| bolts.lastError            | String                                          | Shows the last error occurred in the bolt                                                |
| bolts.errorLapsedSecs      | Integer                                         | Number of seconds elapsed since that last error happened in a bolt                       |
| bolts.errorWorkerLogLink   | String                                          | Link to the worker log that reported the exception                                       |
| bolts.emitted              | Long                                            | Number of tuples emitted                                                                 |
| bolts.requestedMemOnHeap   | Double                                          | Requested On-Heap Memory by                                                              |

|                           |        | User (MB)                              |
|---------------------------|--------|----------------------------------------|
| bolts.requestedMemOffHeap | Double | Requested Off-Heap Memory by User (MB) |
| bolts.requestedCpu        | Double | Requested CPU by User (%)              |

## Examples:

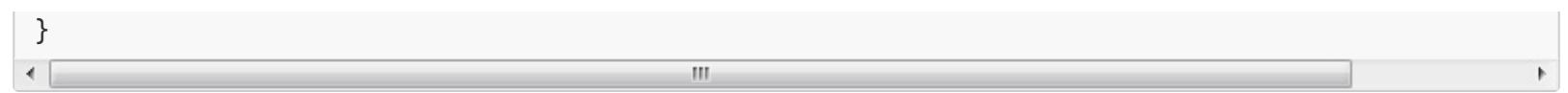
1. <http://ui-daemon-host-name:8080/api/v1/topology/WordCount3-1-1402960825>
2. <http://ui-daemon-host-name:8080/api/v1/topology/WordCount3-1-1402960825?sys=1>
3. <http://ui-daemon-host-name:8080/api/v1/topology/WordCount3-1-1402960825>window=600>

## Sample response:

```
{
  "name": "WordCount3",
  "id": "WordCount3-1-1402960825",
  "workersTotal": 3,
  "window": "600",
  "status": "ACTIVE",
  "tasksTotal": 28,
  "executorsTotal": 28,
  "uptime": "29m 19s",
  "uptimeSeconds": 1759,
  "msgTimeout": 30,
  "windowHint": "10m 0s",
  "schedulerDisplayResource": true,
  "topologyStats": [
    {
      "windowPretty": "10m 0s",
      "window": "600",
      "emitted": 397960,
      "transferred": 213380,
      "completeLatency": "0.000",
      "acked": 213460,
      "failed": 0
    },
    {
      "windowPretty": "3h 0m 0s",
      "window": "10800",
      "emitted": 1190260,
      "transferred": 638260,
      "completeLatency": "0.000",
      "acked": 638280,
      "failed": 0
    },
    {
      "windowPretty": "1d 0h 0m 0s",
      "window": "86400",
      "emitted": 1190260,
      "transferred": 638260,
      "completeLatency": "0.000",
      "acked": 638280,
      "failed": 0
    }
  ]
}
```

```
        "failed": 0
    },
    {
        "windowPretty": "All time",
        "window": ":all-time",
        "emitted": 1190260,
        "transferred": 638260,
        "completeLatency": "0.000",
        "acked": 638280,
        "failed": 0
    }
],
"workers": [
    {
        "topologyName": "WordCount3",
        "topologyId": "WordCount3-1-1402960825",
        "host": "192.168.10.237",
        "supervisorId": "bdfe8eff-f1d8-4bce-81f5-9d3ae1bf432e-169.254.129.212",
        "uptime": "2m 47s",
        "uptimeSeconds": 167,
        "port": 6707,
        "workerLogLink": "http://192.168.10.237:8000/log?file=WordCount3-1-1402960825%2F6707%2Fwo",
        "componentNumTasks": {
            "spout": 5
        },
        "executorsTotal": 8,
        "assignedMemOnHeap": 704.0,
        "assignedCpu": 130.0,
        "assignedMemOffHeap": 80.0
    }
],
"spouts": [
    {
        "executors": 5,
        "emitted": 28880,
        "completeLatency": "0.000",
        "transferred": 28880,
        "acked": 0,
        "spoutId": "spout",
        "tasks": 5,
        "lastError": "",
        "errorLapsedSecs": null,
        "failed": 0
    }
],
"bolts": [
    {
        "executors": 12,
        "emitted": 184580,
        "transferred": 0,
        "acked": 184640,
        "executeLatency": "0.048",
        "tasks": 12,
        "executed": 184620,
        "processLatency": "0.043",
        "boltId": "count",
        "lastError": "",
        "errorLapsedSecs": null,
        "capacity": "0.003",
        "failed": 0
    }
]
```

```
},
{
  "executors": 8,
  "emitted": 184500,
  "transferred": 184500,
  "acked": 28820,
  "executeLatency": "0.024",
  "tasks": 8,
  "executed": 28780,
  "processLatency": "2.112",
  "boltId": "split",
  "lastError": "",
  "errorLapsedSecs": null,
  "capacity": "0.000",
  "failed": 0
}
],
"configuration": {
  "storm.id": "WordCount3-1-1402960825",
  "dev.zookeeper.path": "/tmp/dev-storm-zookeeper",
  "topology.tick.tuple.freq.secs": null,
  "topology.builtin.metrics.bucket.size.secs": 60,
  "topology.fall.back.on.java.serialization": true,
  "topology.max.error.report.per.interval": 5,
  "zmq.linger.millis": 5000,
  "topology.skip.missing.kryo.registrations": false,
  "storm.messaging.netty.client_worker_threads": 1,
  "ui.childopts": "-Xmx768m",
  "storm.zookeeper.session.timeout": 20000,
  "nimbus.reassign": true,
  "topology.trident.batch.emit.interval.millis": 500,
  "storm.messaging.netty.flush.check.interval.ms": 10,
  "nimbus.monitor.freq.secs": 10,
  "logviewer.childopts": "-Xmx128m",
  "java.library.path": "/usr/local/lib:/opt/local/lib:/usr/lib",
  "topology.executor.send.buffer.size": 1024,
  "storm.local.dir": "storm-local",
  "storm.messaging.netty.buffer_size": 5242880,
  "supervisor.worker.start.timeout.secs": 120,
  "topology.enable.message.timeouts": true,
  "nimbus.cleanup.inbox.freq.secs": 600,
  "nimbus.inbox.jar.expiration.secs": 3600,
  "drpc.worker.threads": 64,
  "topology.worker.shared.thread.pool.size": 4,
  "nimbus.seeds": [
    "hw10843.local"
],
"storm.messaging.netty.min_wait_ms": 100,
"storm.zookeeper.port": 2181,
"transactional.zookeeper.port": null,
"topology.executor.receive.buffer.size": 1024,
"transactional.zookeeper.servers": null,
"storm.zookeeper.root": "/storm",
"storm.zookeeper.retry.intervalceiling.millis": 30000,
"supervisor.enable": true,
"storm.messaging.netty.server_worker_threads": 1
},
"replicationCount": 1
```



## /api/v1/topology/:id/metrics

Returns detailed metrics for topology. It shows metrics per component, which are aggregated by stream.

| Parameter | Value                                  | Description                                       |
|-----------|----------------------------------------|---------------------------------------------------|
| id        | String (required)                      | Topology Id                                       |
| window    | String. Default value :all-time        | window duration for metrics in seconds            |
| sys       | String. Values 1 or 0. Default value 0 | Controls including sys stats part of the response |

Response fields:

| Field                        | Value                             | Description                                                   |
|------------------------------|-----------------------------------|---------------------------------------------------------------|
| window                       | String. Default value ":all-time" | window duration for metrics in seconds                        |
|                              | windowHint                        | String                                                        |
| spouts                       | Array                             | Array of all the spout components in the topology             |
| spouts.id                    | String                            | Spout id                                                      |
| spouts.emitted               | Array                             | Array of all the output streams this spout emits messages     |
| spouts.emitted.stream_id     | String                            | Stream id for this stream                                     |
| spouts.emitted.value         | Long                              | Number of messages emitted in given window                    |
| spouts.transferred           | Array                             | Array of all the output streams this spout transfers messages |
| spouts.transferred.stream_id | String                            | Stream id for this stream                                     |

|                                  |                                                 |                                                                      |
|----------------------------------|-------------------------------------------------|----------------------------------------------------------------------|
| spouts.transferred.value         | Long                                            | Number messages transferred in given window                          |
| spouts.acked                     | Array                                           | Array of all the output streams this spout receives ack of messages  |
| spouts.acked.stream_id           | String                                          | Stream id for this stream                                            |
| spouts.acked.value               | Long                                            | Number of messages acked in given window                             |
| spouts.failed                    | Array                                           | Array of all the output streams this spout receives fail of messages |
| spouts.failed.stream_id          | String                                          | Stream id for this stream                                            |
| spouts.failed.value              | Long                                            | Number of messages failed in given window                            |
| spouts.complete_ms_avg           | Array                                           | Array of all the output streams this spout receives ack of messages  |
| spouts.complete_ms_avg.stream_id | String                                          | Stream id for this stream                                            |
| spouts.complete_ms_avg.value     | String (double value returned in String format) | Total latency for processing the message                             |
| bolts                            | Array                                           | Array of all the bolt components in the topology                     |
| bolts.id                         | String                                          | Bolt id                                                              |
| bolts.emitted                    | Array                                           | Array of all the output streams this bolt emits messages             |
| bolts.emitted.stream_id          | String                                          | Stream id for this stream                                            |
| bolts.emitted.value              | Long                                            | Number of messages emitted in given window                           |
|                                  |                                                 | Array of all the output                                              |

|                                   |                                                 |                                                                    |
|-----------------------------------|-------------------------------------------------|--------------------------------------------------------------------|
| bolts.transferred                 | Array                                           | streams this bolt transfers messages                               |
| bolts.transferred.stream_id       | String                                          | Stream id for this stream                                          |
| bolts.transferred.value           | Long                                            | Number messages transferred in given window                        |
| bolts.acked                       | Array                                           | Array of all the input streams this bolt acknowledges of messages  |
| bolts.acked.component_id          | String                                          | Component id for this stream                                       |
| bolts.acked.stream_id             | String                                          | Stream id for this stream                                          |
| bolts.acked.value                 | Long                                            | Number of messages acked in given window                           |
| bolts.failed                      | Array                                           | Array of all the input streams this bolt receives fail of messages |
| bolts.failed.component_id         | String                                          | Component id for this stream                                       |
| bolts.failed.stream_id            | String                                          | Stream id for this stream                                          |
| bolts.failed.value                | Long                                            | Number of messages failed in given window                          |
| bolts.process_ms_avg              | Array                                           | Array of all the input streams this spout acks messages            |
| bolts.process_ms_avg.component_id | String                                          | Component id for this stream                                       |
| bolts.process_ms_avg.stream_id    | String                                          | Stream id for this stream                                          |
| bolts.process_ms_avg.value        | String (double value returned in String format) | Average time of the bolt to ack a message after it was received    |

|                                                 |                                                 |                                                                     |
|-------------------------------------------------|-------------------------------------------------|---------------------------------------------------------------------|
| <code>bolts.executed</code>                     | Array                                           | Array of all the input streams this bolt executes messages          |
| <code>bolts.executed.component_id</code>        | String                                          | Component id for this stream                                        |
| <code>bolts.executed.stream_id</code>           | String                                          | Stream id for this stream                                           |
| <code>bolts.executed.value</code>               | Long                                            | Number of messages executed in given window                         |
| <code>bolts.executed_ms_avg</code>              | Array                                           | Array of all the output streams this spout receives ack of messages |
| <code>bolts.executed_ms_avg.component_id</code> | String                                          | Component id for this stream                                        |
| <code>bolts.executed_ms_avg.stream_id</code>    | String                                          | Stream id for this stream                                           |
| <code>bolts.executed_ms_avg.value</code>        | String (double value returned in String format) | Average time to run the execute method of the bolt                  |

## Examples:

- ```

1. http://ui-daemon-host-name:8080/api/v1/topology/WordCount3-1-1402960825/metrics
1. http://ui-daemon-host-name:8080/api/v1/topology/WordCount3-1-1402960825/metrics?sys=1
2. http://ui-daemon-host-name:8080/api/v1/topology/WordCount3-1-1402960825/metrics?window=600

```

## Sample response:

```
{
  "window":":all-time",
  "window-hint":"All time",
  "spouts":[
    {
      "id":"spout",
      "emitted":[
        {
          "stream_id":"__metrics",
          "value":20
        },
        {
          "stream_id":"default",
          "value":17350280
        },
        {
          "stream_id":"__ack_init",
        }
      ]
    }
  ]
}
```

```
        "value":17328160
    },
    {
        "stream_id":"__system",
        "value":20
    }
],
"transferred":[
    {
        "stream_id":"__metrics",
        "value":20
    },
    {
        "stream_id":"default",
        "value":17350280
    },
    {
        "stream_id":"__ack_init",
        "value":17328160
    },
    {
        "stream_id":"__system",
        "value":0
    }
],
"acked":[
    {
        "stream_id":"default",
        "value":17339180
    }
],
"failed":[
],
"complete_ms_avg":[
    {
        "stream_id":"default",
        "value":"920.497"
    }
]
},
"bolts":[
{
    "id":"count",
    "emitted":[
        {
            "stream_id":"__metrics",
            "value":120
        },
        {
            "stream_id":"default",
            "value":190748180
        },
        {
            "stream_id":"__ack_ack",
            "value":190718100
        }
    ]
}]]
```

```
{  
    "stream_id": "__system",  
    "value": 20  
}  
],  
"transferred": [  
    {  
        "stream_id": "__metrics",  
        "value": 120  
    },  
    {  
        "stream_id": "default",  
        "value": 0  
    },  
    {  
        "stream_id": "__ack_ack",  
        "value": 190718100  
    },  
    {  
        "stream_id": "__system",  
        "value": 0  
    }  
],  
"acked": [  
    {  
        "component_id": "split",  
        "stream_id": "default",  
        "value": 190733160  
    }  
],  
"failed": [  
],  
"process_ms_avg": [  
    {  
        "component_id": "split",  
        "stream_id": "default",  
        "value": "0.004"  
    }  
],  
"executed": [  
    {  
        "component_id": "split",  
        "stream_id": "default",  
        "value": 190733140  
    }  
],  
"executed_ms_avg": [  
    {  
        "component_id": "split",  
        "stream_id": "default",  
        "value": "0.005"  
    }  
]  
},  
{  
    "id": "split",  
    "emitted": [  
]
```

```
{  
    "stream_id":"__metrics",  
    "value":60  
},  
{  
    "stream_id":"default",  
    "value":190754740  
},  
{  
    "stream_id":"__ack_ack",  
    "value":17317580  
},  
{  
    "stream_id":"__system",  
    "value":20  
}  
],  
"transferred": [  
    {  
        "stream_id":"__metrics",  
        "value":60  
    },  
    {  
        "stream_id":"default",  
        "value":190754740  
    },  
    {  
        "stream_id":"__ack_ack",  
        "value":17317580  
    },  
    {  
        "stream_id":"__system",  
        "value":0  
    }  
],  
"acked": [  
    {  
        "component_id": "spout",  
        "stream_id": "default",  
        "value": 17339180  
    }  
],  
"failed": [  
],  
"process_ms_avg": [  
    {  
        "component_id": "spout",  
        "stream_id": "default",  
        "value": "0.051"  
    }  
],  
"executed": [  
    {  
        "component_id": "spout",  
        "stream_id": "default",  
        "value": 17339240  
    }  
]
```

```

        ],
        "executed_ms_avg": [
            {
                "component_id": "spout",
                "stream_id": "default",
                "value": "0.052"
            }
        ]
    }
}

```

## /api/v1/topology/:id/component/:component (GET)

Returns detailed metrics and executor information

Parameter	Value	Description
id	String (required)	Topology Id
component	String (required)	Component Id
window	String. Default value :all-time	window duration for metrics in seconds
sys	String. Values 1 or 0. Default value 0	controls including sys stats part of the response

Response fields:

Field	Value	Description
user	String	Topology owner
id	String	Component id
encodedId	String	URL encoded component id
name	String	Topology name
executors	Integer	Number of executor tasks in the component
tasks	Integer	Number of instances of component
requestedMemOnHeap	Double	Requested On-Heap Memory by User (MB)
requestedMemOffHeap	Double	Requested Off-Heap

		Memory by User (MB)
requestedCpu	Double	Requested CPU by User (%)
schedulerDisplayResource	Boolean	Whether to display scheduler resource information
topologyId	String	Topology id
topologyStatus	String	Topology status
encodedTopologyId	String	URL encoded topology id
window	String. Default value "All Time"	window duration for metrics in seconds
componentType	String	component type: SPOUT or BOLT
windowHint	String	window param value in "hh mm ss" format. Default value is "All Time"
debug	Boolean	If debug is enabled for the component
samplingPct	Double	Controls downsampling of events before they are sent to event log (percentage)
eventLogLink	String	URL viewer link to event log (debug mode)
profilingAndDebuggingCapable	Boolean	true if there is support for Profiling and Debugging Actions
profileActionEnabled	Boolean	true if worker profiling (Java Flight Recorder) is enabled
profilerActive	Array	Array of currently active Profiler Actions
componentErrors	Array of Errors	List of component errors

componentErrors.errorTime	Long	Timestamp when the exception occurred (Prior to 0.11.0, this field was named 'time'.)
componentErrors.errorHost	String	host name for the error
componentErrors.errorPort	String	port for the error
componentErrors.error	String	Shows the error happened in a component
componentErrors.errorLapsedSecs	Integer	Number of seconds elapsed since the error happened in a component
componentErrors.errorWorkerLogLink	String	Link to the worker log that reported the exception
spoutSummary	Array	(only for spouts) Array of component stats, one element per window.
spoutSummary.windowPretty	String	Duration passed in HH:MM:SS format
spoutSummary.window	String	window duration for metrics in seconds
spoutSummary.emitted	Long	Number of messages emitted in given window
spoutSummary.completeLatency	String (double value returned in String format)	Total latency for processing the message
spoutSummary.transferred	Long	Total number of messages transferred in given window
spoutSummary.acked	Long	Number of messages acked
spoutSummary.failed	Long	Number of messages failed
boltStats	Array	(only for bolts) Array of component stats, one element per window.

boltStats.windowPretty	String	Duration passed in HH:MM:SS format
boltStats.window	String	window duration for metrics in seconds
boltStats.transferred	Long	Total number of messages transferred in given window
boltStats.processLatency	String (double value returned in String format)	Average time of the bolt to ack a message after it was received
boltStats.acked	Long	Number of messages acked
boltStats.failed	Long	Number of messages failed
inputStats	Array	(only for bolts) Array of input stats
inputStats.component	String	Component id
inputStats.encodedComponentId	String	URL encoded component id
inputStats.executeLatency	Long	The average time a tuple spends in the execute method
inputStats.processLatency	Long	The average time it takes to ack a tuple after it is first received
inputStats.executed	Long	The number of incoming tuples processed
inputStats.acked	Long	Number of messages acked
inputStats.failed	Long	Number of messages failed
inputStats.stream	String	The name of the tuple stream given in the topology, or "default" if none specified
outputStats	Array	Array of output stats
		Number of tuples emitted

outputStats.transferred	Long	that sent to one or more bolts
outputStats.emitted	Long	Number of tuples emitted
outputStats.stream	String	The name of the tuple stream given in the topology, or "default" if none specified

## Examples:

- ```
1. http://ui-daemon-host-name:8080/api/v1/topology/WordCount3-1-1402960825/component/spout
2. http://ui-daemon-host-name:8080/api/v1/topology/WordCount3-1-1402960825/component/spout?sys=1
3. http://ui-daemon-host-name:8080/api/v1/topology/WordCount3-1-1402960825/component/spout?window=6
```

## Sample response:

```
{
  "name": "WordCount3",
  "id": "spout",
  "componentType": "spout",
  "windowHint": "10m 0s",
  "executors": 5,
  "componentErrors": [{"errorTime": 1406006074000,
                      "errorHost": "10.11.1.70",
                      "errorPort": 6701,
                      "errorWorkerLogLink": "http://10.11.1.7:8000/log?file=worker-6701.log",
                      "errorLapsedSecs": 16,
                      "error": "java.lang.RuntimeException: java.lang.StringIndexOutOfBoundsException"}],
  "topologyId": "WordCount3-1-1402960825",
  "tasks": 5,
  "window": "600",
  "profilerActive": [
    {
      "host": "10.11.1.70",
      "port": "6701",
      "dumplink": "http://10.11.1.70:8000/dumps/ex-1-1452718803/10.11.1.70%3A6701",
      "timestamp": "576328"
    }
  ],
  "profilingAndDebuggingCapable": true,
  "profileActionEnabled": true,
  "spoutSummary": [
    {
      "windowPretty": "10m 0s",
      "window": "600",
      "emitted": 28500,
      "transferred": 28460,
      "completeLatency": "0.000",
      "acked": 0,
      "failed": 0
    }
  ]
}
```

```
},
{
  "windowPretty": "3h 0m 0s",
  "window": "10800",
  "emitted": 127640,
  "transferred": 127440,
  "completeLatency": "0.000",
  "acked": 0,
  "failed": 0
},
{
  "windowPretty": "1d 0h 0m 0s",
  "window": "86400",
  "emitted": 127640,
  "transferred": 127440,
  "completeLatency": "0.000",
  "acked": 0,
  "failed": 0
},
{
  "windowPretty": "All time",
  "window": ":all-time",
  "emitted": 127640,
  "transferred": 127440,
  "completeLatency": "0.000",
  "acked": 0,
  "failed": 0
}
],
"outputStats": [
  {
    "stream": "__metrics",
    "emitted": 40,
    "transferred": 0,
    "completeLatency": "0",
    "acked": 0,
    "failed": 0
  },
  {
    "stream": "default",
    "emitted": 28460,
    "transferred": 28460,
    "completeLatency": "0",
    "acked": 0,
    "failed": 0
  }
],
"executorStats": [
  {
    "workerLogLink": "http://10.11.1.7:8000/log?file=worker-6701.log",
    "emitted": 5720,
    "port": 6701,
    "completeLatency": "0.000",
    "transferred": 5720,
    "host": "10.11.1.7",
    "acked": 0,
    "uptime": "43m 4s",
    "uptimeSeconds": 2584,
```

```
"id": "[24-24]",
"failed": 0
},
{
  "workerLogLink": "http://10.11.1.7:8000/log?file=worker-6703.log",
  "emitted": 5700,
  "port": 6703,
  "completeLatency": "0.000",
  "transferred": 5700,
  "host": "10.11.1.7",
  "acked": 0,
  "uptime": "42m 57s",
  "uptimeSeconds": 2577,
  "id": "[25-25]",
  "failed": 0
},
{
  "workerLogLink": "http://10.11.1.7:8000/log?file=worker-6702.log",
  "emitted": 5700,
  "port": 6702,
  "completeLatency": "0.000",
  "transferred": 5680,
  "host": "10.11.1.7",
  "acked": 0,
  "uptime": "42m 57s",
  "uptimeSeconds": 2577,
  "id": "[26-26]",
  "failed": 0
},
{
  "workerLogLink": "http://10.11.1.7:8000/log?file=worker-6701.log",
  "emitted": 5700,
  "port": 6701,
  "completeLatency": "0.000",
  "transferred": 5680,
  "host": "10.11.1.7",
  "acked": 0,
  "uptime": "43m 4s",
  "uptimeSeconds": 2584,
  "id": "[27-27]",
  "failed": 0
},
{
  "workerLogLink": "http://10.11.1.7:8000/log?file=worker-6703.log",
  "emitted": 5680,
  "port": 6703,
  "completeLatency": "0.000",
  "transferred": 5680,
  "host": "10.11.1.7",
  "acked": 0,
  "uptime": "42m 57s",
  "uptimeSeconds": 2577,
  "id": "[28-28]",
  "failed": 0
}
]
```

# Profiling and Debugging GET Operations

## /api/v1/topology/:id/profiling/start/:host-port/:timeout (GET)

Request to start profiler on worker with timeout. Returns status and link to profiler artifacts for worker.

| Parameter | Value             | Description                              |
|-----------|-------------------|------------------------------------------|
| id        | String (required) | Topology Id                              |
| host-port | String (required) | Worker Id                                |
| timeout   | String (required) | Time out for profiler to stop in minutes |

Response fields:

| Field    | Value  | Description                                          |
|----------|--------|------------------------------------------------------|
| id       | String | Worker id                                            |
| status   | String | Response Status                                      |
| timeout  | String | Requested timeout                                    |
| dumplink | String | Link to logviewer URL for worker profiler documents. |

Examples:

1. http://ui-daemon-host-name:8080/api/v1/topology/wordcount-1-1446614150/profiling/start/10.11.1.7
2. http://ui-daemon-host-name:8080/api/v1/topology/wordcount-1-1446614150/profiling/start/10.11.1.7
3. http://ui-daemon-host-name:8080/api/v1/topology/wordcount-1-1446614150/profiling/start/10.11.1.7

Sample response:

```
{  
  "status": "ok",  
  "id": "10.11.1.7:6701",  
  "timeout": "10",  
  "dumplink": "http://10.11.1.7:8000/dumps/wordcount-1-1446614150/10.11.1.7%3A6701"  
}
```

## /api/v1/topology/:id/profiling/dumpprofile/:host-port (GET)

Request to dump profiler recording on worker. Returns status and worker id for the request.

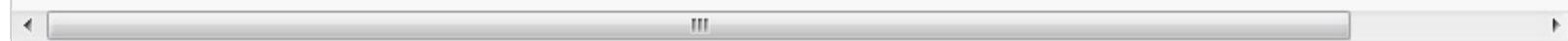
| Parameter | Value             | Description |
|-----------|-------------------|-------------|
| id        | String (required) | Topology Id |
| host-port | String (required) | Worker Id   |

Response fields:

| Field  | Value  | Description     |
|--------|--------|-----------------|
| id     | String | Worker id       |
| status | String | Response Status |

Examples:

1. <http://ui-daemon-host-name:8080/api/v1/topology/wordcount-1-1446614150/profiling/dumpprofile/10>.



Sample response:

```
{  
  "status": "ok",  
  "id": "10.11.1.7:6701",  
}
```

## /api/v1/topology/:id/profiling/stop/:host-port (GET)

Request to stop profiler on worker. Returns status and worker id for the request.

| Parameter | Value             | Description |
|-----------|-------------------|-------------|
| id        | String (required) | Topology Id |
| host-port | String (required) | Worker Id   |

Response fields:

| Field | Value  | Description |
|-------|--------|-------------|
| id    | String | Worker id   |

|        |        |                 |
|--------|--------|-----------------|
| status | String | Response Status |
|--------|--------|-----------------|

Examples:

1. <http://ui-daemon-host-name:8080/api/v1/topology/wordcount-1-1446614150/profiling/stop/10.11.1.7>

Sample response:

```
{
  "status": "ok",
  "id": "10.11.1.7:6701",
}
```

### /api/v1/topology/:id/profiling/dumpjstack/:host-port (GET)

Request to dump jstack on worker. Returns status and worker id for the request.

| Parameter | Value             | Description |
|-----------|-------------------|-------------|
| id        | String (required) | Topology Id |
| host-port | String (required) | Worker Id   |

Response fields:

| Field  | Value  | Description     |
|--------|--------|-----------------|
| id     | String | Worker id       |
| status | String | Response Status |

Examples:

1. <http://ui-daemon-host-name:8080/api/v1/topology/wordcount-1-1446614150/profiling/dumpjstack/10.11.1.7>

Sample response:

```
{
  "status": "ok",
  "id": "10.11.1.7:6701",
}
```

### /api/v1/topology/:id/profiling/dumpheap/:host-port (GET)

Request to dump heap (jmap) on worker. Returns status and worker id for the request.

| Parameter | Value             | Description |
|-----------|-------------------|-------------|
| id        | String (required) | Topology Id |
| host-port | String (required) | Worker Id   |

Response fields:

| Field  | Value  | Description     |
|--------|--------|-----------------|
| id     | String | Worker id       |
| status | String | Response Status |

Examples:

```
1. http://ui-daemon-host-name:8080/api/v1/topology/wordcount-1-1446614150/profiling/dumpheap/10.11.
```

Sample response:

```
{
  "status": "ok",
  "id": "10.11.1.7:6701",
}
```

## /api/v1/topology/:id/profiling/restartworker/:host-port (GET)

Request to request the worker. Returns status and worker id for the request.

| Parameter | Value             | Description |
|-----------|-------------------|-------------|
| id        | String (required) | Topology Id |
| host-port | String (required) | Worker Id   |

Response fields:

| Field  | Value  | Description     |
|--------|--------|-----------------|
| id     | String | Worker id       |
| status | String | Response Status |

## Examples:

```
1. http://ui-daemon-host-name:8080/api/v1/topology/wordcount-1-1446614150/profiling/restartworker/1
```

## Sample response:

```
{  
    "status": "ok",  
    "id": "10.11.1.7:6701",  
}
```

## POST Operations

### /api/v1/topology/:id/activate (POST)

Activates a topology.

| Parameter | Value             | Description |
|-----------|-------------------|-------------|
| id        | String (required) | Topology Id |

## Sample Response:

```
{"topologyOperation": "activate", "topologyId": "wordcount-1-1420308665", "status": "success"}
```

### /api/v1/topology/:id/deactivate (POST)

Deactivates a topology.

| Parameter | Value             | Description |
|-----------|-------------------|-------------|
| id        | String (required) | Topology Id |

## Sample Response:

```
{"topologyOperation": "deactivate", "topologyId": "wordcount-1-1420308665", "status": "success"}
```

### /api/v1/topology/:id/rebalance/:wait-time (POST)

Rebalances a topology.

| Parameter | Value | Description |
|-----------|-------|-------------|
|           |       |             |

|                         |                   |                                    |
|-------------------------|-------------------|------------------------------------|
| <b>id</b>               | String (required) | Topology Id                        |
| <b>wait-time</b>        | String (required) | Wait time before rebalance happens |
| <b>rebalanceOptions</b> | Json (optional)   | topology rebalance options         |

Sample rebalanceOptions json:

```
{"rebalanceOptions" : {"numWorkers" : 2, "executors" : {"spout" : 4, "count" : 10}}, "callback" : "f
```

Examples:

```
curl -i -b ~/cookiejar.txt -c ~/cookiejar.txt -X POST  
-H "Content-Type: application/json"  
-d '{"rebalanceOptions": {"numWorkers": 2, "executors": { "spout" : "5", "split": 7, "count": 5 }}}  
http://localhost:8080/api/v1/topology/wordcount-1-1420308665/rebalance/0
```

Sample Response:

```
{"topologyOperation":"rebalance","topologyId":"wordcount-1-1420308665","status":"success"}
```

## /api/v1/topology/:id/kill/:wait-time (POST)

Kills a topology.

| Parameter        | Value             | Description                        |
|------------------|-------------------|------------------------------------|
| <b>id</b>        | String (required) | Topology Id                        |
| <b>wait-time</b> | String (required) | Wait time before rebalance happens |

Caution: Small wait times (0-5 seconds) may increase the probability of triggering the bug reported in [STORM-112](#), which may result in broker Supervisor daemons.

Sample Response:

```
{"topologyOperation":"kill","topologyId":"wordcount-1-1420308665","status":"success"}
```

## API errors

The API returns 500 HTTP status codes in case of any errors.

Sample response:

```
{  
  "error": "Internal Server Error",  
  "errorMessage": "java.lang.NullPointerException\\n\\tat clojure.core$name.invoke(core.clj:1505)\\n\\t  
}  

```

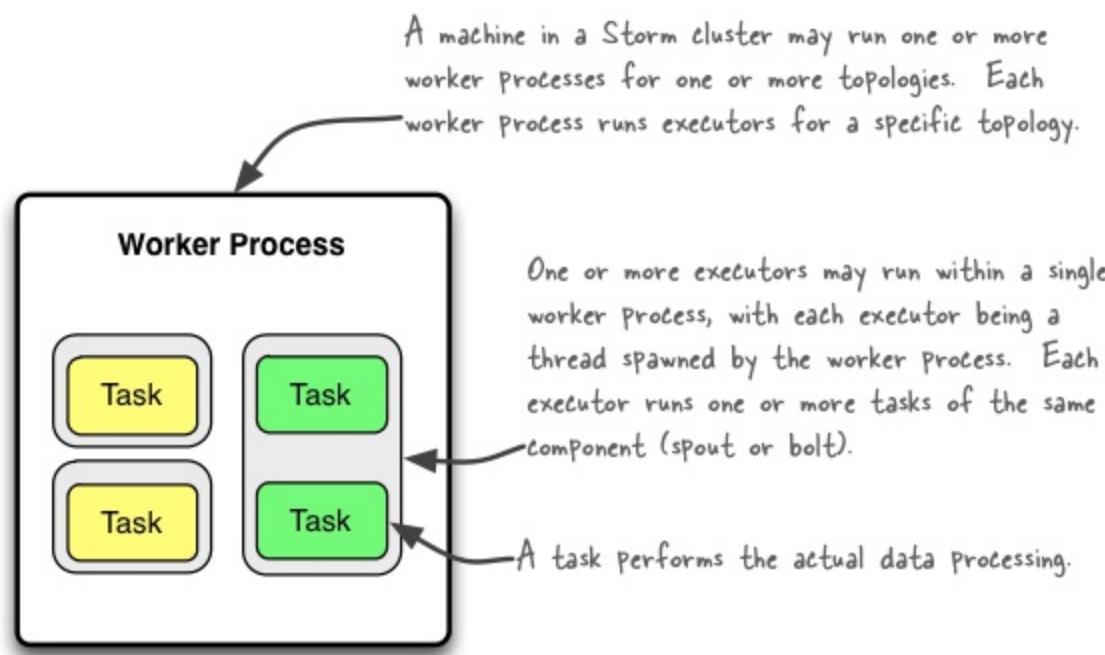
# 理解 Storm Topology 的 Parallelism (并行度)

什么让 **topology** (拓扑) 可以运行: **worker** 进程, **executors** (执行器) 和 **tasks** (任务)

Storm 区分以下 3 个主要的实体, 它们在 Storm 集群中用于实际的运行 topology (拓扑) :

1. Worker 进程
2. Executors (线程)
3. Tasks

这是一个简单的例子, 以说明他们之间的关系



一个 **worker** 进程 执行一个 **topology** (拓扑) 的子集. 一个 **worker** 进程属于一个指定 **topology** (拓扑), 并且针对该 **topology** 的一个或多个组件 (**spouts** 或 **bolts**) 来说会运行一个或更多的 **executors** (执行器). 一个正在运行的 **topology** 由许多这样的进程组成, 它们运行在 Storm 集群的多个机器上.

一个 **executor** (执行器) 是一个线程, 它是由 **worker** 进程产生的. 它可能针对相同的组件 (**spout** 或 **blot**) 运行一个或多个 **tasks** (任务) .

一个 **task** 执行实际的数据处理 - 在您代码中实现的每个 **spout** 或 **bolt** 在整个集群上都执行了许多的 **taskk** (任务). 组件的 **task** (任务) 数量在 **topology** (拓扑) 的整个生命周期中总是相同的, 但组件的 **executors** (线程) 数量可能会随时间而变化. 这意味着以下条件成立:

`#threads ≤ #tasks`. 默认情况下, 默认情况下, **tasks** (任务) 数量与 **executors** (执行器) 设置

成一样, 例如. Storm 在每个线程上运行一个 task (任务) .

## 配置 topology 的 parallelism (并行度)

请注意, 在 Storm 的术语中, "parallelism (并行度)" 特别用于描述所谓的 *parallelism hint*, 它表示组件的 executor (线程) 的初始化数量. 在本文档中, 虽然我们在一般意义上使用术语 "parallelism (并行度)" 来描述如何配置的不仅只有 executor (执行器) 的数量, 还可以配置 worker 进程的数量以及 Storm topology 的 tasks (任务) 数量. We will specifically call out when "parallelism" is used in the normal, narrow definition of Storm.

以下部分概述了各种配置选项以及如何在代码中设置它们. 尽管设置这些选项有多种方法, 表中只列出了其中的一些选项. Storm 目前有以下 配置设置的优先顺序: `defaults.yaml` < `storm.yaml` < topology 级别指定的配置 < 内部 component (组件) 指定的配置 < 外部 component (组件) 指定的配置.

### worker 进程的数量

- 描述: 在集群的机器中有多少个 worker 进程来 针对 topology 以创建它.
- 配置选项: [TOPOLOGY\\_WORKERS](#)
- 如何在代码中设置 (示例) :
  - [Config#setNumWorkers](#)

### executors (线程) 的数量

- 描述: 每个 component (组件) 产生多少个 executors (执行器) .
- 配置选项: `None` (传递 `parallelism_hint` 参数到 `setSpout` 或 `setBolt`)
- 如何在代码中设置 (示例) :
  - [TopologyBuilder#setSpout\(\)](#)
  - [TopologyBuilder#setBolt\(\)](#)
  - 参数现在指定了 bolt 的 executors (执行器) 的初始化数量 (不是 tasks) .

### tasks (任务) 的数量

- 描述: per component 有多少个 tasks (任务) 来创建他们.
- 配置选项: [TOPOLOGY\\_TASKS](#)
- 如何在代码中设置 (示例) :
  - [ComponentConfigurationDeclarer#setNumTasks\(\)](#)

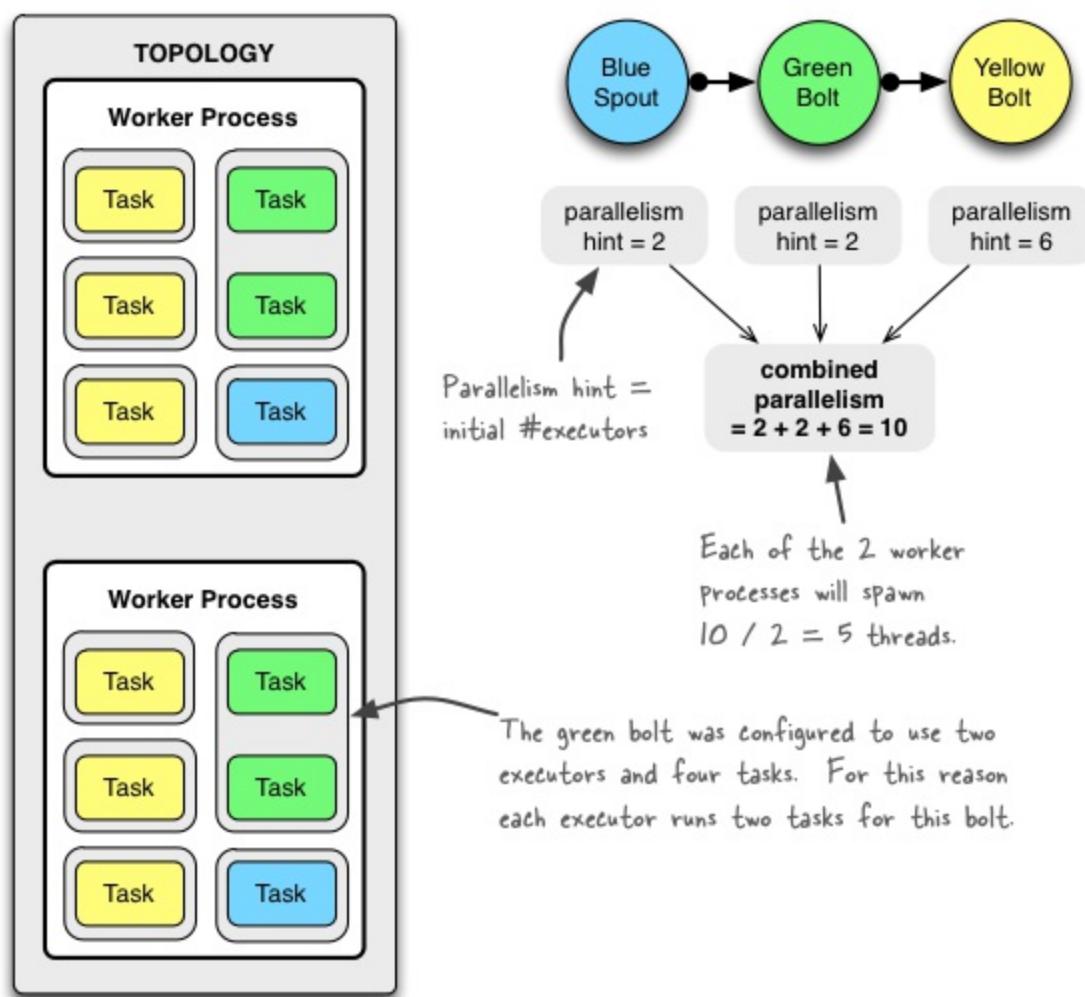
以下是在练习中显示这些设置的示例代码片段:

```
topologyBuilder.setBolt("green-bolt", new GreenBolt(), 2)
    .setNumTasks(4)
    .shuffleGrouping("blue-spout");
```

在上面的代码中，我们配置了 Storm 来运行 Bolt "GreenBolt"，其初始数量为两个 executor（执行器）和四个相关联的 tasks（任务）。Storm 的每个 executor（线程）将会运行两个 tasks（任务）。如果您不显示的配置 tasks 的数量，Storm 将使用每个 executor 一个 task 的默认配置来运行它们。

## 运行 topology 的示例

下图显示了简单的 topology（拓扑）是如何运行的。该 topology 由 3 个 components（组件）构成：一个名为 BlueSpout 的 spout 和两个名为 GreenBolt 和 YellowBolt 的 bolts。该组件链接，使得 BlueSpout 将其输出发送到 GreenBolt，它们将自己的输出发送到 YellowBolt。



该 GreenBolt 按照上面的代码片段进行配置，而 BlueSpout 和 YellowBolt 只设置了 parallelism hint（执行器数量）。以下是相关代码：

```

Config conf = new Config();
conf.setNumWorkers(2); // use two worker processes

topologyBuilder.setSpout("blue-spout", new BlueSpout(), 2); // set parallelism hint to 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()
);

```

当然, Storm 还提供了额外的配置设置来控制 **topology** (拓扑) 的并行度, 包括:

- **TOPOLOGY\_MAX\_TASK\_PARALLELISM**: 此设置针对单个组件生成的 **executor** (执行器) 的数量设置上限. 通常在测试期间使用它来限制在本地模式下运行 **topology** (拓扑) 时产生的线程数. 您可以通过 [Config#setMaxTaskParallelism\(\)](#) 来设置此选项.

## 如何改变正在运行中的 **topology** 的并行度

Storm 的一个很好的特性是可以增加或减少 **worker** 进程 和/或 **executor** (执行器) 的数量, 而无需重新启动集群或 **topology** (拓扑). 这样做的行为称之为 **rebalancing** (重新平衡).

您有两个选项来 **rebalance** (重新平衡) 一个 **topology** (拓扑) :

1. 使用 **Storm web UI** 来 **rebalance** (重新平衡) 指定的 **topology**.
2. 使用 **CLI** 工具 **storm rebalance**, 如下所示.

以下是一个使用 **CLI** 工具的示例:

```

## 重新配置 topology "mytopology" 以使用 5 个 worker 进程,
## 该 spout "blue-spout" 要使用 3 个 executors (执行器) 并且
## 该 bolt "yellow-bolt" 要使用 10 executors (执行器) .

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

```

## 参考文献

- [概念](#)
- [配置](#)

- 在生产集群上运行 `topologies` (拓扑) ]
- Local mode (本地模式)
- 教程
- Storm API 文档, most notably the class `Config`

# FAQ

---

## 最佳实践

针对配置 **Storm+Trident**, 您可以给我哪些经验呢?

- worker 的数量是机器数量的倍数; 并行度是 worker 数量的倍数; kafka partitions 的数量是 spout 并行度的倍数
- 每个机器上的每个 topology 使用一个 worker
- Start with fewer, larger aggregators, one per machine with workers on it
- 使用 isolation scheduler (隔离调度器)
- 每个 worker 使用一个 acker -- 0.9 版本默认是这样的, 但是更早的版本没有这样.
- 启用 GC 日志记录; 在正常情况下, 你应该看到很少的 major GC.
- set the trident batch millis to about 50% of your typical end-to-end latency.
- Start with a max spout pending that is for sure too small -- one for trident, or the number of executors for storm -- and increase it until you stop seeing changes in the flow. You'll probably end up with something near  $2 * (\text{throughput in recs/sec}) * (\text{end-to-end latency})$  (2x the Little's law capacity).

**What are some of the best ways to get a worker to mysteriously and bafflingly die?**

什么是获取

- 您是否有对 log directory (日志目录) 的写入权限
- 您扩大过你的 heap 大小吗?
- 是否所有的 workers 都安装了正确的 libraries (函数库) ?
- 是否 zookeeper 的 hostname (主机名) 仍然设置为 localhost 了?
- 您提供了一个正确, 唯一的 hostname (主机名) -- 它可以解析回机器上 -- 对于每个 worker, 并且将它们放入 storm conf 文件中?
- 您是否双向开启了 firewall/securitygroup 的权限 a) 所有的 workers, b) storm master, c) zookeeper? 另外, 从 workers 到您的 topology 访问的任何 kafka/kestrel/database/etc ? 使用 netcat 来检测下对应的 ports (端口) 并且确定一下.

**Help!** 我不能看到:

- **my logs** 默认情况下, 日志为 \$STORM\_HOME/logs. 检查您是否具有该目录的写入权限.

他们配置在 `log4j2/{cluster, worker}.xml` 文件中.

- **final JVM settings** 在 `childopts` 中添加 `-XX+PrintFlagsFinal` 命令行选项（请看配置文件）
- **final Java system properties** 添加

`Properties props = System.getProperties(); props.list(System.out);` 靠近您构建 topology（拓扑）的地方.

## 我应该使用多少个 **Workers**？

worker 的数量是由 `supervisors` 来确定的 -- 每个 `supervisor` 将监督一些 `JVM slots`. 您在 topology（拓扑）上设置的事情是它将尝试声明多少个 `worker slots`.

每台机器每个 topology（拓扑）使用多个 `worker` 没有很好的理由。

一个 topology（拓扑）运行在三个 8 核心的节点上, 并行度是 24, 每台机器的每个 bolt 将得到 8 个 executor（执行器）, 即每个核心一个. 与运行三个 worker（每个有 8 个指定的 executor）相比, 有 24 个 worker（每个分配一个 executor）的运行有 3 个大的优势.

第一, 对同一个 worker 的 executor 进行重新分区（shuffles 或 group-bys）的数据不必放入传输缓冲区. 相反, tuple 直接从发送到接收缓冲区存储. 这是一个很大的优势. 相反, 如果目标 executor 在不同 worker 的同一台计算机上, 则必须执行 `send -> worker transfer -> local socket -> worker recv -> exec recv buffer`. 它不经过打网卡, 但并不像 executor 在同一个 worker 那么大.

通常情况下, 三个具有非常大的 `backing cache`（后备缓存）的 `aggregator`（聚合器）比拥有小的 `backing caches`（后台缓存）的二十四个 `aggregators`（聚合器）更好, 因为这样减少了数据倾斜的影响, 并提高了 LRU 效率.

最后, 更少的 `workers` 降低了控制 `flow` 的难度.

## Topology（拓扑）

### 一个 Trident topology 可以有多个 Streams 吗？

Trident topology 可以像带条件路径（if-else）的 workflow（工作流）一样工作吗? 例如. 一个 Spout(S1) 连接到 bolt(b0), 其基于进入 tuple（元组）中的某些值将它们引导到 bolt (B1) 或 bolt (B2) , 而不是两者都有.

一个 Trident 的 "each" 操作返回一个 Stream 对象, 你可以在一个变量中存储它. 然后, 您可

以在同一个 Stream 上运行多个 each 进行 split 拆分, 例如:

```
Stream s = topology.each(...).groupBy(...).aggregate(...)  
Stream branch1 = s.each(..., FilterA)  
Stream branch2 = s.each(..., FilterB)
```

你可以使用 join, merge 或 multiReduce 来 join streams.

在写入操作时, 您不能向 Trident 的 emit (发射) 多个输出流 -- 请参阅 [STORM-68](#)

当我启动 **topology** 时, 为什么获得一个 **NotSerializableException/IllegalStateException** 异常?

在 Storm 的生命周期中, 在执行 topology 之前, topology 被实例化, 然后序列化为字节格式以存储在 ZooKeeper 中. 在此步骤中, 如果 topology 中的 Spout 或 Bolt 具有初始化的不可序列化属性, 序列化将会失败. 如果需要一个不序列化的字段, 请在将 topology 传递给 worker 之后运行的 blot 或 spout 的 prepare 方法中进行初始化.

## Spouts

**coordinator** 是什么, 为什么有几个?

trident-spout 实际运行在 storm bolt 之内. trident topology 的 storm-spout 是 MasterBatchCoordinator -- 它协调了 trident batches, 无论您使用什么 spout 都是一样的. 当 MBC 为每个 spout-coordinators 分配一个 seed tuple (种子元组) 时, batch 就诞生了. spout-coordinator bolts 知道您特定的 spouts 应该如何配合 -- 所以在 kafka 的场景中, 这有助于找出每个 spout 应该从哪个 partition 和 offset 进行 pull 操作.

## What can I store into the spout's metadata record?

You should only store static data, and as little of it as possible, into the metadata record (note: maybe you *can* store more interesting things; you shouldn't, though)

'emitPartitionBatchNew' 函数多久被调用一次?

Since the MBC is the actual spout, all the tuples in a batch are just members of its tupletree. That means storm's "max spout pending" config effectively defines the number of concurrent batches trident runs. The MBC emits a new batch if it has fewer than max-spending tuples pending and if at least one [trident batch interval](#)'s worth of seconds has

passed since the last batch.

由于 MBC 是实际的 spout，所以一个 batch 中的所有 tuple 只是它的 tupletree 的成员。这意味着 storm 的 "max spout pending" 配置有效地定义了并发 batch trident 运行的次数。

## If nothing was emitted does Trident slow down the calls?

Yes, there's a pluggable "spout wait strategy"; the default is to sleep for a [configurable amount of time](#)

OK, 那么 **trident batch** 间隔是多少？

你知道 486 时代的电脑有一个 [turbo button](#) 吗？

Actually, it has two practical uses. One is to throttle spouts that poll a remote source without throttling processing. For example, we have a spout that looks in a given S3 bucket for a new batch-uploaded file to read, linebreak and emit. We don't want it hitting S3 more than every few seconds: files don't show up more than once every few minutes, and a batch takes a few seconds to process.

The other is to limit overpressure on the internal queues during startup or under a heavy burst load -- if the spouts spring to life and suddenly jam ten batches' worth of records into the system, you could have a mass of less-urgent tuples from batch 7 clog up the transfer buffer and prevent the \$commit tuple from batch 3 to get through (or even just the regular old tuples from batch 3). What we do is set the trident batch interval to about half the typical end-to-end processing latency -- if it takes 600ms to process a batch, it's OK to only kick off a batch every 300ms.

Note that this is a cap, not an additional delay -- with a period of 300ms, if your batch takes 258ms Trident will only delay an additional 42ms.

你是怎样设置 **batch** 大小的？

Trident 不对 batch 数量设置自己的限制。在 Kafka spout 的场景中，最大抓取的字节大小初一平均的记录大小定义了每个子分区的有效记录。

如何调整 **batch** 的大小？

trident batch 是一个有点过载的设施。与 partition（分区）数量一起，batch 大小受限于或用

于定义:

1. the unit of transactional safety (tuples at risk vs time)
2. per partition, an effective windowing mechanism for windowed stream analytics
3. per partition, the number of simultaneous queries that will be made by a partitionQuery, partitionPersist, etc;
4. per partition, the number of records convenient for the spout to dispatch at the same time;

一旦生成，您将无法更改总体的 **batch** 大小，但您可以更改 **partition** 数量 - 执行 **shuffle**, 然后更改 **parallelism hint** (并行度)

## Time Series (时间序列)

如何按时间聚合事件？

如果您的记录具有不可变的 **timestamp** (时间戳)，并且您想 **count**, **average** 或以其他方式将它们聚合到离散时间段中，则 **Trident** 是一款出色且可扩展的解决方案。

编写一个将 **timestamp** 转换成 **time bucket** 的 **Each** 函数: 如果 **bucket** 的大小是 "by hour (按小时的)"，则时间戳 `2013-08-08 12:34:56` 将被映射成 `2013-08-08 12:00:00` **time bucket**, 十二点钟以后的其它时间也是这样. 然后在那个 **timebucket** 上的 **group** (组) 并使用分组的 **persistentAggregate** 方法. **persistentAggregate** 使用由数据存储支持的本地 **cacheMap**. 具有许多记录的 **Groups** 需要从数据存储器读取很少的数据，并使用高效的批量读写. 只要您的数据供给相对较快，**Trident** 就可以非常有效地利用内存和网络. 即使服务器脱机一天，然后在一瞬间提供全天的数据，旧的结果将被安静地检索和更新 -- 并且不干扰计算当前结果.

我怎么知道一个时间内的 **bucket** 的所有记录已经被收到？

你不能知道所有的 **event** (事件) 都被收集 -- 这是一个 **epistemological challenge** (认识论的挑战)，而不是分布式系统的挑战. 您可以:

- 使用 **domain knowledge** 设置时间限制
- 引入 **punctuation**: 一个 **record** 知道紧跟特定时间 **bucket** 内所有记录之后而来. **Trident** 使用此方案知道 **batch** 何时完成. 例如，如果您从一组传感器接收记录，则每个传感器都将按照传感器的顺序发送，所有传感器都会向您发送 `3: 02: xx` 或更后版本的时间戳，以让您知道可以 **commit** (提交) .

- 在可能的情况下, 使您的进程增加: 进来的每个 `value` 会让答案越来越正确. `Trident ReducerAggregator` 是一个 `operator`, 它采取先前的结果和一组新的记录, 并返回一个新的结果. 这样可以将结果缓存并序列化到数据存储; 如果一台服务器脱机一天, 然后在一天内回来一整天的数据, 旧的结果将被平静地检索和更新.
- `Lambda` 架构: 在接收时将所有 `event` (时间) 记录到 `archival store` (`S3, HBase, HDFS`). 在快速处理的层面上, 一旦时间窗口被 `clear` (清楚), 处理 `bucket` 以获得可行的答案, 并忽略比时间窗口更旧的一切. 定期运行全局聚合以计算 "正确的" 答案。

# **Layers on Top of Storm**

# Storm Trident

# Trident 教程

Trident 是在 Storm 基础上, 一个以实时计算为目标的 **high-level abstraction** (高度抽象) . 它在提供处理大吞吐量数据能力 (每秒百万次消息) 的同时, 也提供了低延时分布式查询和 **stateful stream processing** (有状态流式处理) 的能力. 如果你对 Pig 和 Cascading 这种高级批处理工具很了解的话, 那么应该很容易理解 Trident , 因为他们之间很多的概念和思想都是类似的. Trident 提供了 joins , aggregations, grouping, functions, 以及 filters 等能力. 除此之外, Trident 还提供了一些专门的 primitives (原语), 从而在基于数据库或者其他存储的前提下应付有状态的递增式处理. Trident 也提供一致性 (**consistent**) 、有且仅有一次 (**exactly-once**) 等语义, 这使得我们在使用 trident topology 时变得容易.

## 举例说明

让我们一起来看一个 Trident 的例子. 在这个例子中, 我们主要做了两件事情:

1. 从一个 **input stream** 中读取语句并计算每个单词的个数
2. 提供查询给定单词列表中每个单词当前总数的功能

为了说明的目的, 本例将从如下这样一个无限的输入流中读取语句作为输入:

```
FixedBatchSpout spout = new FixedBatchSpout(new Fields("sentence"), 3,
    new Values("the cow jumped over the moon"),
    new Values("the man went to the store and bought some candy"),
    new Values("four score and seven years ago"),
    new Values("how many apples can you eat"));
spout.setCycle(true);
```

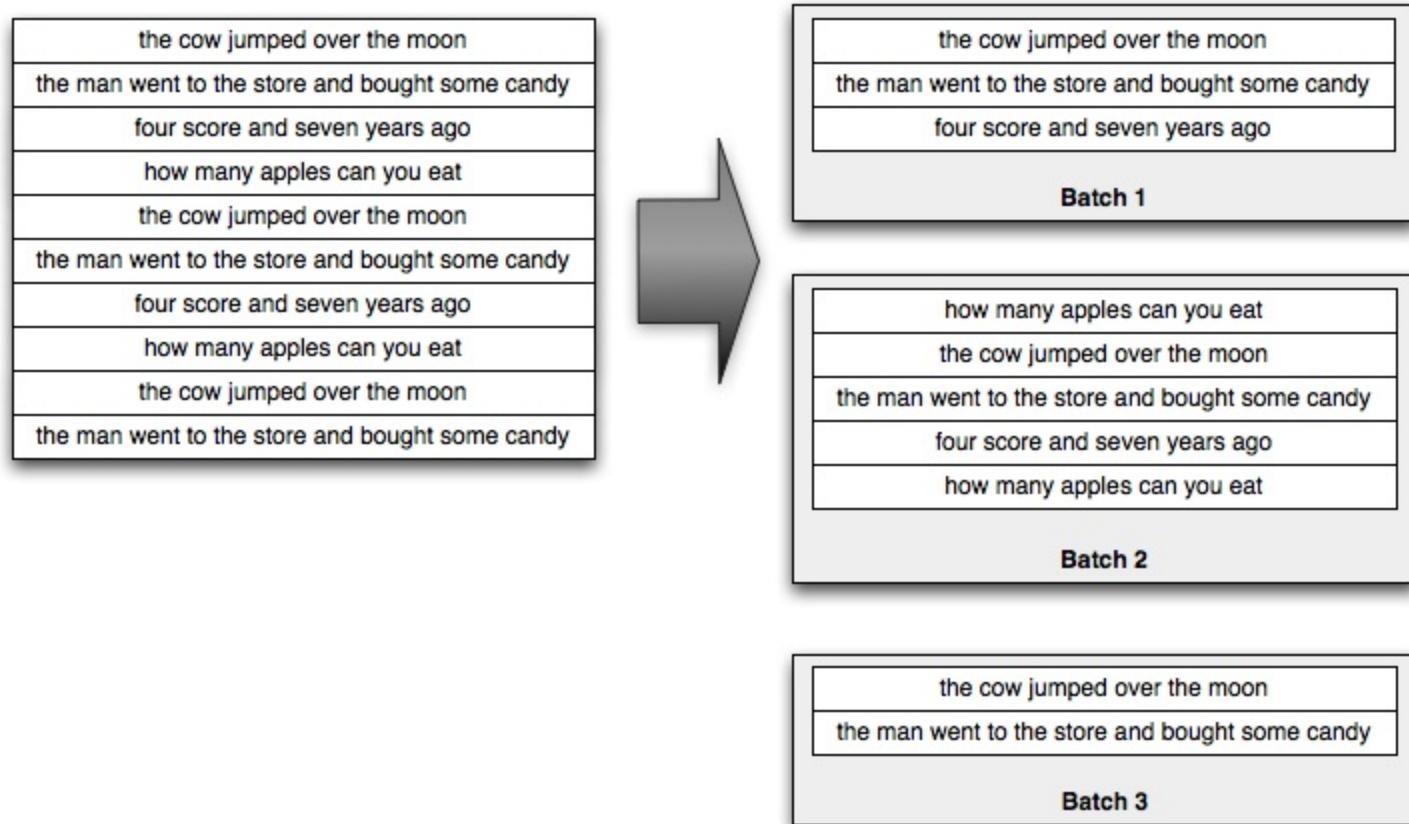
这个 spout 会循环 sentences 数据集, 不断输出 sentence stream , 下面的代码会以这个 stream 作为输入并计算每个单词的个数

```
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"))
        .parallelismHint(6);
```

在这段代码中, 我们首先创建了一个 TridentTopology 对象, 该对象提供了相应的接口去 constructing Trident computations (构造 Trident 计算过程) . TridentTopology 类中的 newStream 方法从 input source (输入源) 中读取数据, 并在 topology 中创建一个新的数据

流. 在这个例子中, 我们使用了上面定义的 `FixedBatchSpout` 对象作为 `input source` (输入源) . `Input sources` (输入数据源) 同样也可以是如 `Kestrel` 或者 `Kafka` 这样的队列代理. `Trident` 会在 `Zookeeper` 中保存每个 `input source` (输入源) 一小部分 `state` 信息 (关于它已经消费的数据的 `metadata` 信息) 来追踪数据的处理情况, "spout1" 字符串指定 `Trident` 应保留 `metadata` 信息到 `Zookeeper` 的哪个节点.

`Trident` 在处理输入 `stream` 的时候会转换成 `batch` (包含若干个 `tuple`) 来处理. 比如说, 输入的 `sentence stream` 可能会被拆分成如下的 `batch` :



一般来说, 这些小的 `batch` 中的 `tuple` 可能在数千或者数百万这样的数量级, 这完全取决于你的 `incoming throughput` (输入的吞吐量) .

`Trident` 提供了一系列非常成熟的批处理 API 来处理这些小 batches . 这些 API 和你在 Pig 或者 Cascading 中看到的非常类似, 你可以做 `groupby`, `join`, `aggregation`, 执行 `function` 和 `filter` 等等. 当然, 独立的处理每个小的 `batch` 并不是非常有趣的事情, 所以 `Trident` 提供了功能来实现 `batch` 之间的聚合, 并可以将这些聚合的结果存储到内存, Memcached, Cassandra 或者是一些其他的存储中. 同时, `Trident` 还提供了非常好的功能来查询这些数据源的实时状态, 这些实时状态可以被 `Trident` 更新, 同时这些状态也可以是一个 `independent source of state` (独立的状态源) .

回到我们的这个例子中来, `spout` 输出了一包含单一字段 "sentence" stream . 在下一行, `topology` 使用了 `Split` 函数将 `stream` 拆分成一个个 `tuple` , `Split` 函数读取 `stream` (输入流) 中的 "sentence" 字段并将其拆分成若干个 word tuple . 每一个 sentence tuple 可能会被转换成多个 word tuple, 比如说 "the cow jumped over the moon" 会被转换成 6 个 "word" tuples. 下面是 `Split` 的定义:

```
public class Split extends BaseFunction {  
    public void execute(TridentTuple tuple, TridentCollector collector) {  
        String sentence = tuple.getString(0);  
        for(String word: sentence.split(" ")) {  
            collector.emit(new Values(word));  
        }  
    }  
}
```

如你所见, 真的很简单. 它只是简单的根据空格拆分 `sentence` , 并将拆分出的每个单词作为一个 `tuple` 输出.

`topology` 计算完成后, 会将计算结果持久化保存. 首先, `word stream` 被根据 "word" 字段进行 `group` 操作, 然后每一个 `group` 使用 `Count` 聚合器进行持久化聚合. `persistentAggregate` 方法会帮助你把一个聚合的结果保存或者更新到状态源中. 在这个例子中, 单词的计数结果保存在内存中, 不过我们可以很简单的把结果保存到其他的存储类型中, 如 `Memcached`, `Cassandra` 等持久化存储. 如果我们要把结果存储到 `Memcached` 中, 只是简单的使用下面这句话替换掉 `persistentAggregate` 这一行就可以 (使用 `trident-memcached` ), 这当中的 "serverLocations" 是 `Memcached cluster` 的 host/ports (主机和端口号) 列表:

```
.persistentAggregate(MemcachedState.transactional(serverLocations), new Count(), new Fields("count")  
MemcachedState.transactional()
```

`persistentAggregate` 存储的数据就是 `stream` 输出的所有 `batches` 聚合的结果.

`Trident` 非常酷的一点就是它提供 `fully fault-tolerant` (完整的容错性), `exactly-once` (处理一次且仅一次) 的语义. 这就让你可以很轻松的使用 `Trident` 来进行实时数据处理. `Trident` 会把状态以某种形式持久化, 以至于错误发生时 `retries is necessary`, 但是 `Trident` 不会对相同的源数据多次 `update` 数据库.

`persistentAggregate` 方法会把数据流转换成一个 `TridentState` 对象. 在这个例子当中, `TridentState` 对象代表了所有的单词计数结果. 我们会使用这个 `TridentState` 对象来实现在计算过程中的分布式查询部分.

上面的是 **topology** 中的第一部分, **topology** 的第二部分实现了一个低延时的单词数量的分布式查询. 这个查询以一个用空格分割的单词列表为输入, 并返回这些单词的总个数. 这些查询就像普通的 RPC 调用那样被执行的, 要说不同的话, 那就是他们在后台是并行执行的. 下面是执行查询的一个例子:

```
DRPCClient client = new DRPCClient("drpc.server.location", 3772);
System.out.println(client.execute("words", "cat dog the man"));
// prints the JSON-encoded result, e.g.: "[[5078]]"
```

如你所见, 除了在 **storm cluster** 上并行执行之外, 这个查询看上去就是一个普通的 RPC 调用. 这样的简单查询的延时通常在 10 毫秒左右. 当然, 更复杂的 DRPC 调用可能会占用更长的时间, 但是延时很大程度上是取决于你给计算分配了多少资源.

**Topology** 中的分布式查询部分实现如下所示:

```
topology.newDRPCStream("words")
    .each(new Fields("args"), new Split(), new Fields("word"))
    .groupBy(new Fields("word"))
    .stateQuery(wordCounts, new Fields("word"), new MapGet(), new Fields("count"))
    .each(new Fields("count"), new FilterNull())
    .aggregate(new Fields("count"), new Sum(), new Fields("sum"));
```

我们仍然是使用 **TridentTopology** 对象来创建 DRPC stream , 并且我们将这个函数命名为 "words" . 这个函数名会作为第一个参数在使用 DRPC Client 来执行查询的时候用到.

每个 DRPC 请求会被当做处理 **little batch** 的 job, 这个 job 输入一个代表请求的单一 **tuple**. 这个 tuple 包含了一个叫做 "args" 的字段, 在这个字段中保存了客户端提供的查询参数. 在这个例子中, 这个参数是一个以空格分割的单词列表.

首先, 我们使用 **Split** 函数把传入的请求参数拆分成独立的单词. 然后按照 **word** 对 **stream** 进行 **group** 操作, 之后就可以使用 **stateQuery** 操作查询 **topology** 在第一部分生成的 **TridentState** 对象. **stateQuery** 接受一个 **source of state** (**state** 源) (在这个例子中, 就是我们的 **topology** 所计算的单词的个数的结果) 以及一个用于查询的函数作为输入. 在这个例子中, 我们使用了 **MapGet** 函数来获取每个单词的出现个数. 由于 DRPC stream 是使用跟 **TridentState** 完全同样的 **group** 方式 (按照 "word" 字段进行 **group by** ), 每个单词的查询会被路由到 **TridentState** 对象的分区, **TridentState** 对象是用来管理和更新 **word** 计数结果的.

接下来, 我们用 **FilterNull** 这个过滤器将没有 **count** 结果的 **words** 过滤掉, 并使用 **Sum** 这个

聚合器将这些 count 累加起来得到结果. 最终, Trident 会自动把这个结果发送回等待的客户端.

Trident 在如何最大程度地保证执行 topology (拓扑) 性能方面是非常智能的. 在 topology 中会自动的发生两件非常有意思的事情:

1. 读取和写入状态的操作 (比如说 stateQuery 和 persistentAggregate ) 会自动地批量处理到该状态. 如果当前处理的 batch 中有 20 次 updates 需要被同步到存储中. Trident 会自动的把这些操作汇总到一起, 只做一次读和一次写 (在许多情况下, 您可以在 State implementation 中使用缓存来减少读请求), 而不是进行 20 次读 20 次写的操作. 因此你可以在很方便的执行计算的同时, 保证了非常好的性能.
2. Trident 的聚合器已经是被优化的非常好的了. Trident 并不是简单的把一个 group 中所有的 tuples 都发送到同一个机器上面进行聚合, 而是在发送之前已经进行过一次部分的聚合. 打个比方, Count 聚合器会先在每个 partition 上面进行 count, 然后把每个分片 count 汇总到一起就得到了最终的 count . 这个技术其实就跟 MapReduce 里面的 combiner 是一个思想.

让我们再来看一下 Trident 的另外一个例子.

## Reach

这个例子是一个纯粹的 DRPC topology , 这个 topology 会计算一个给定 URL 的 reach 值, reach 值是该 URL 对应页面的推文能够 Reach (送达) 的用户数量, 那么我们就把这个数量叫做这个 URL 的 reach . 要计算 reach , 你需要获取转发过这个推文的所有人, 然后找到所有该转发者的粉丝, 并将这些粉丝去重, 最后就得到了去重后的用户的数量. 如果把计算 reach 的整个过程都放在一台机器上面, 就太困难了, 因为这会需要数千次数据库调用以及千万级别数量的 tuple . 如果使用 Storm 和 Trident , 你就可以把这些计算步骤在整个 cluster 中并行进行 (具体哪些步骤, 可以参考 DRPC 介绍一文, 该文有介绍过 Reach 值的计算方法) .

这个 topology 会读取两个 sources of state (state 源) : 这是两个 map 集合, 一个将该 URL 映射到所有转发该推文的用户列表, 还有一个将用户映射到该用户的粉丝列表. topology 的定义如下:

```
TridentState urlToTweeters =
    topology.newStaticState(getUrlToTweetersState());
TridentState tweetersToFollowers =
    topology.newStaticState(getTweeterToFollowersState());
```

```
topology.newDRPCStream("reach")
    .stateQuery(urlToTweeters, new Fields("args"), new MapGet(), new Fields("tweeters"))
    .each(new Fields("tweeters"), new ExpandList(), new Fields("tweeter"))
    .shuffle()
    .stateQuery(tweetersToFollowers, new Fields("tweeter"), new MapGet(), new Fields("followers"))
    .parallelismHint(200)
    .each(new Fields("followers"), new ExpandList(), new Fields("follower"))
    .groupBy(new Fields("follower"))
    .aggregate(new One(), new Fields("one"))
    .parallelismHint(20)
    .aggregate(new Count(), new Fields("reach"));
```

这个 topology 使用 newStaticState 方法创建了 TridentState 对象来代表一个外部数据库. 使用这个 TridentState 对象, 我们就可以在这个 topology 上面进行动态查询了. 和所有的 sources of state 一样, 在这些数据库上面的查找会自动被批量执行, 从而最大程度的提升效率.

这个 topology 的定义是非常简单的 – 它仅是一个 simple batch processing job (简单的批处理的作业). 首先, 查询 urlToTweeters 数据库来得到转发过这个 URL 的用户列表. 这个查询会返回一个 tweeter 列表, 因此我们使用 ExpandList 函数来把其中的每一个 tweeter 转换成一个 tuple .

接下来, 我们来获取每个 tweeter 的 follower . 我们使用 shuffle 来把要处理的 tweeter 均匀地分配到 topology 运行的每一个 worker 中并发去处理. 然后查询 tweetersToFollowers 数据库从而的到每个转发者的 followers. 你可以看到我们为 topology 的这部分分配了很大的并行度, 这是因为这部分是整个 topology 中最耗资源的计算部分.

然后, 我们对这些粉丝进行去重和计数. 这分为如下两步:首先, 通过 "follower" 字段对流进行 "group by" 分组, 并对每个 group 执行 "One" 聚合器. "One" 聚合器对每个 group 简单的发送一个 tuple, 该 tuple 仅包含一个数字 "1". 然后, 将这些 "1" 加到一起, 得到去重后的粉丝集中的粉丝数. "One" 聚合器的定义如下:

```
public class One implements CombinerAggregator<Integer> {
    public Integer init(TridentTuple tuple) {
        return 1;
    }

    public Integer combine(Integer val1, Integer val2) {
        return 1;
    }

    public Integer zero() {
        return 1;
    }
}
```

```
}
```

这是一个 "combiner aggregator" (汇总聚合器) ", 它会在传送结果到其他 worker 汇总之前进行局部汇总, 从而使性能最优. 同样, **Sum** 被定义成一个汇总聚合器, 在 **topology** 的最后部分进行全局求和是高效的.

接下来让我们一起来看看 **Trident** 的一些细节.

## Fields and tuples

**Trident** 的数据模型是 **TridentTuple**, 它是一个 **list**. 在一个 **topology** 中, **tuple** 是在一系列的 **operation** (操作) 中增量生成的. **operation** 一般以一组字段作为输入并输出一组 **function fields** (功能字段). **Operation** 的输入字段经常是输入 **tuple** 的一个子集, 而功能字段则是 **operation** 的输出.

看下面这个例子. 假定你有一个叫做 "**stream**" 的 **stream**, 它包含了 "x", "y" 和 "z" 三个字段. 为了运行一个读取 "y" 作为输入的 **MyFilter** 过滤器, 你可以这样写:

```
stream.each(new Fields("y"), new MyFilter())
```

假设 **MyFilter** 的实现是这样的:

```
public class MyFilter extends BaseFilter {
    public boolean isKeep(TridentTuple tuple) {
        return tuple.getInteger(0) < 10;
    }
}
```

这会保留所有 "y" 字段小于 10 的 **tuples**. 传给 **MyFilter** 的 **TridentTuple** 输入将只包含字段 "y". 这里需要注意的是, 当选择输入字段时, **Trident** 只发送 **tuple** 的一个子集, 这个操作是非常高效的.

让我们一起看一下 "function fields" (功能字段) " 是怎样工作的. 假定你有如下这个函数:

```
public class AddAndMultiply extends BaseFunction {
    public void execute(TridentTuple tuple, TridentCollector collector) {
        int i1 = tuple.getInteger(0);
        int i2 = tuple.getInteger(1);
        collector.emit(new Values(i1 + i2, i1 * i2));
    }
}
```

这个函数接收两个数作为输入并输出两个新的值: 这两个数的和与乘积. 假定你有一个

`stream`, 其中包含 "x", "y" 和 "z" 三个字段. 你可以这样使用这个函数:

```
stream.each(new Fields("x", "y"), new AddAndMultiply(), new Fields("added", "multiplied"));
```

输出的功能字段被添加到输入 `tuple` 后面, 因此这个时候, 每个 `tuple` 中将会有5个字段 "x", "y", "z", "added", 和 "multiplied". "added" 和 "multiplied" 对应于 `AddAndMultiply` 输出的第一个和第二个字段.

另外, 我们可以使用聚合器来用输出字段来替换输入 `tuple`. 如果你有一个 `stream` 包含字段 "val1" 和 "val2", 你可以这样做:

```
stream.aggregate(new Fields("val2"), new Sum(), new Fields("sum"))
```

输出流将会仅包含一个 `tuple`, 该 `tuple` 有一个 "sum" 字段, 这个 `sum` 字段就是一批 `tuple` 中 "val2" 字段的累积和.

但是若对 `group by` 之后的流进行该聚合操作, 则输出 `tuple` 中包含分组字段和聚合器输出的字段, 例如:

```
stream.groupBy(new Fields("val1"))
    .aggregate(new Fields("val2"), new Sum(), new Fields("sum"))
```

这个例子中的输出包含 "val1" 字段和 "sum" 字段.

## State

在实时计算领域的一个主要问题就是怎么样来管理状态, 在面对错误和重试的时候, 更新是幂等的. 消除错误的是不可能的, 当一个节点死掉, 或者一些其他的问题出现时, 这些 batch 需要被重新处理. 问题是-你怎样做状态更新 (无论是外部数据库还是 `topology` 内部的 `State`) 来保证每一个消息被处理且只被处理一次?

这是一个很棘手的问题, 我们可以用接下来的例子进一步说明. 假定你在做一个你的 `stream` 的计数聚合, 并且你想要存储运行时的 `count` 到一个数据库中去. 如果你只是存储这个 `count` 到数据库中, 并且想要进行一次更新, 我们是没有办法知道同样的状态是不是以前已经被 `update`过了的. 这次更新可能在之前就尝试过, 并且已经成功的更新到了数据库中, 不过在后续的步骤中失败了. 还有可能是在上次更新数据库的过程中失败的, 这些你都不知道.

`Trident` 通过做下面两件事情来解决这个问题:

1. 每一个 `batch` 被赋予一个唯一标识 `id` "transaction id". 如果一个 `batch` 被重试, 它将会拥

有和之前同样的 transaction id .

2. State updates (状态更新) 是按照 batch 的顺序进行的 (强顺序). 也就是说, batch 3 的状态更新必须等到 batch 2 的状态更新成功之后才可以进行.

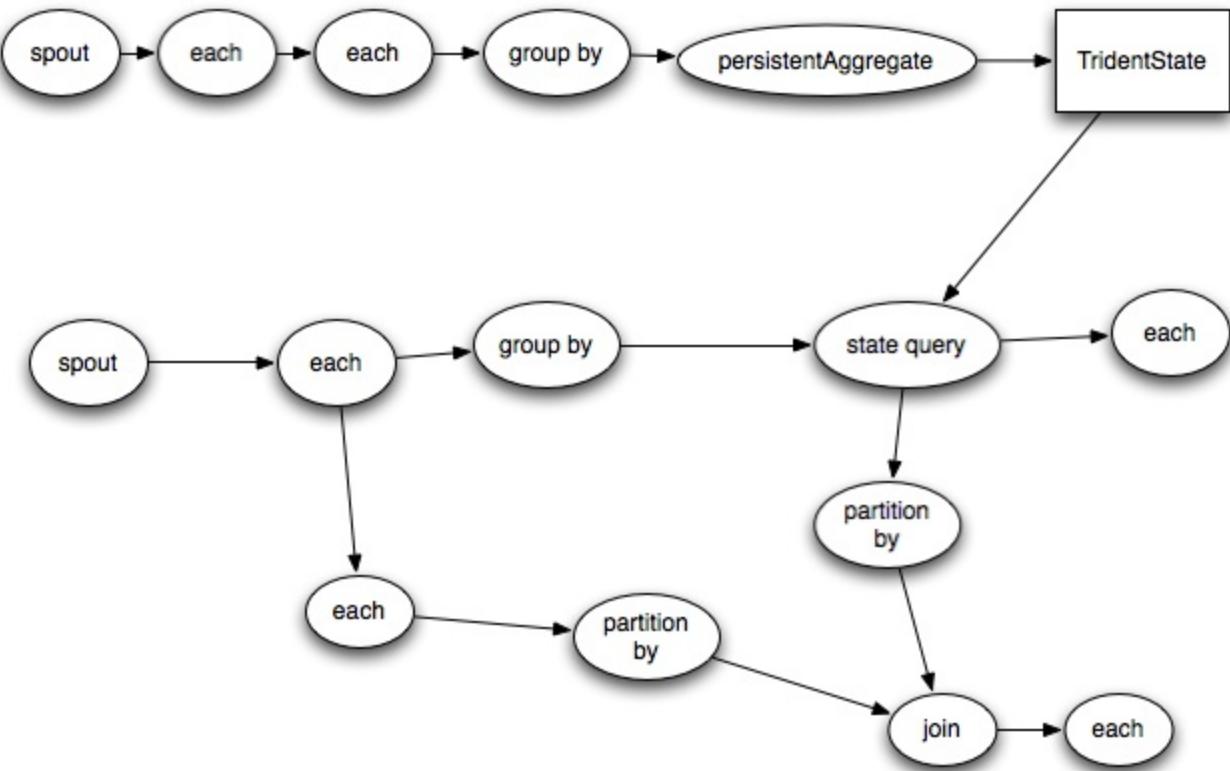
有了这 2 个原则, 你就可以达到有且只有一次更新的目标. 此时, 不是只将 count 存到数据库中, 而是将 transaction id 和 count 作为原子值存到数据库中. 当更新一个 count 的时候, 需要比较数据库中 transaction id 和当前 batch 的 transaction id . 如果相同, 就跳过这次更新. 如果不同, 就更新这个 count .

当然, 你不需要在 topology 中手动处理这些逻辑, 这些逻辑已经被封装在 State 的抽象中并自动进行. 你的 State object 也不需要自己去实现 transaction id 的跟踪操作. 如果你想了解更多的关于如何实现一个 State 以及在容错过程中的一些取舍问题, 可以参照 [这个文档](#).

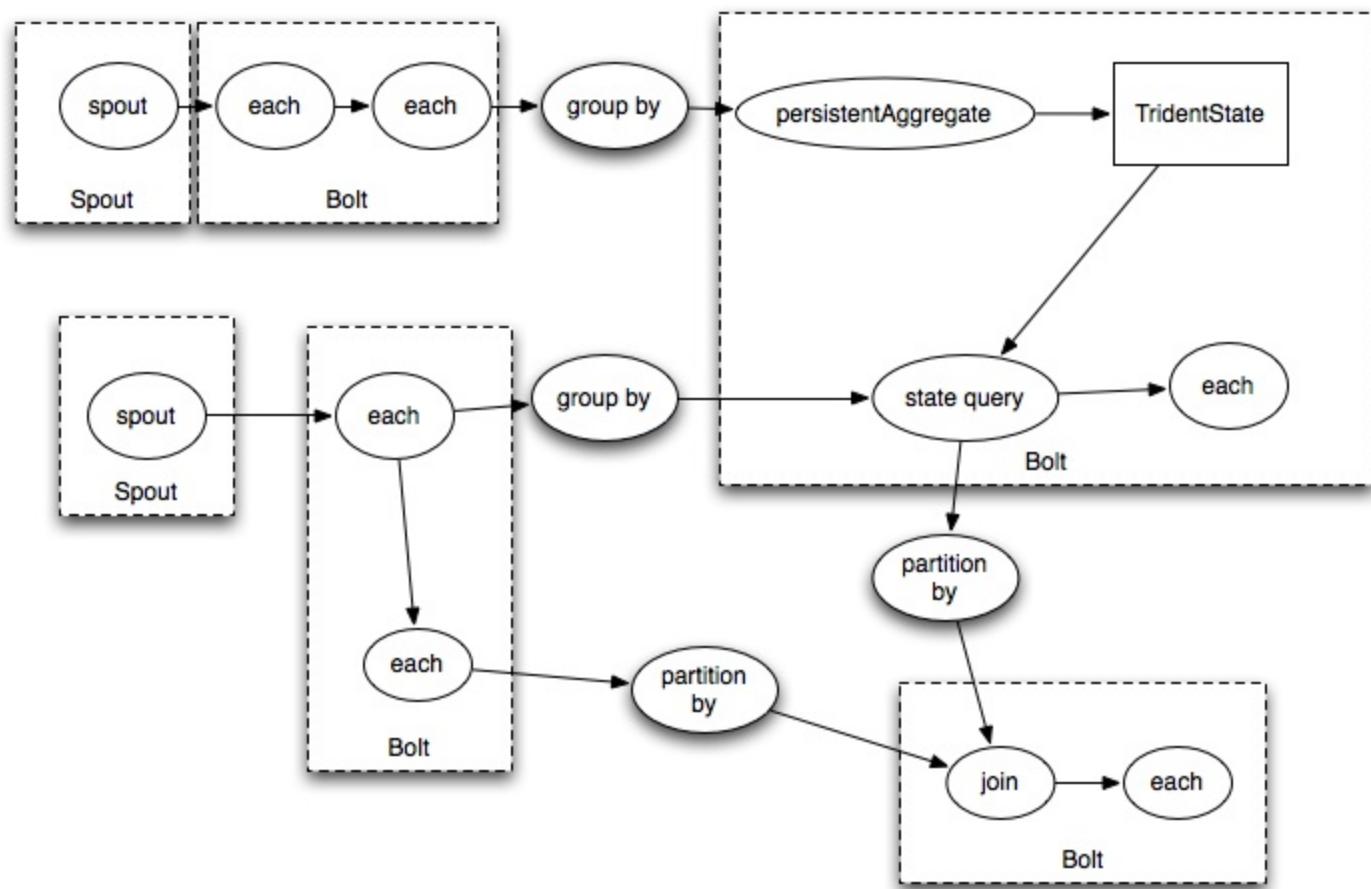
一个 State 可以采用任何策略来存储状态, 它可以存储到一个外部的数据库, 也可以在内存中保持状态并备份到 HDFS 中. State 并不需要永久的保持状态. 比如说, 你有一个内存版的 State 实现, 它保存最近 X 个小时的数据并丢弃老的数据. 可以把 [Memcached integration](#) 作为例子来看看 State 的实现.

## Trident topologies 的执行

Trident 的 topology 会被编译成尽可能高效的 Storm topology . 只有在需要对数据进行 repartition (重新分配) 的时候 (如 groupby 或者 shuffle ) 才会把 tuple 通过 network 发送出去, 如果你有一个 trident topology 如下:



它将会被编译成如下的 Storm spouts/bolts:



小结

**Trident** 使得实时计算更加优雅. 你已经看到了如何使用 **Trident** 的 API 来完成大吞吐量的流式计算, 状态维护, 低延时查询等等功能. **Trident** 让你在获取最大性能的同时, 以更自然的一种方式进行实时计算.

# Trident API 综述

"Stream" 是 Trident 中的核心数据模型, 它被当做一系列的 batch 来处理. 在 Storm 集群的节点之间, 一个 stream 被划分成很多 partition (分区), 对流的 operation (操作) 是在每个 partition 上并行进行的.

注: 1. "Stream" 是 Trident 中的核心数据模型:有些地方也说是 TridentTuple , 没有个标准的说法. 2. 一个 stream 被划分成很多 partition : partition 是 stream 的一个子集, 里面可能有很多 batch , 一个 batch 也可能位于不同的 partition 上.

Trident 有 5 类操作:

1. Partition-local operations , 对每个 partition 的局部操作, 不产生网络传输
2. Repartitioning operations: 对 stream (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. 作为 operation (操作) 的一部分进行网络传输的 Aggregation operations (聚合操作).
4. Operations on grouped streams (作用在分组流上的操作)
5. Merges 和 joins 操作

## Partition-local operations

Partition-local operations (分区本地操作) 不涉及网络传输, 并且独立地应用于每个 batch partition (批处理分区) .

## Functions

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 filter (过滤), 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 :

```
public class MyFunction extends BaseFunction {  
    public void execute(TridentTuple tuple, TridentCollector collector) {  
        for(int i=0; i < tuple.getInteger(0); i++) {  
            collector.emit(new Values(i));  
        }  
    }  
}
```

假设有个叫 "mystream" 的 stream (流)，该流中有如下 tuple ( tuple 的字段为["a", "b", "c"] )：

```
[1, 2, 3]  
[4, 1, 6]  
[3, 0, 8]
```

如果您运行下面的代码：

```
mystream.each(new Fields("b"), new MyFunction(), new Fields("d"))
```

则 resulting tuples (输出 tuple) 中的字段为 ["a", "b", "c", "d"], 如下所示：

```
[1, 2, 3, 0]  
[1, 2, 3, 1]  
[4, 1, 6, 0]
```

## Filters

Filters 收到一个输入 tuple，并决定是否保留该 tuple。假设您拥有这个 filters：

```
public class MyFilter extends BaseFilter {  
    public boolean isKeep(TridentTuple tuple) {  
        return tuple.getInteger(0) == 1 && tuple.getInteger(1) == 2;  
    }  
}
```

现在，假设您有如下这些 tuple，包含字段 ["a", "b", "c"]：

```
[1, 2, 3]  
[2, 1, 1]  
[2, 3, 4]
```

如果您运行如下代码：

```
mystream.filter(new MyFilter())
```

则得到的 resulting tuples (结果 tuples) 为：

```
[1, 2, 3]
```

## map and flatMap

map 返回一个 stream，它包含将给定的 mapping function (映射函数) 应用到 stream 的 tuples 的结果。这个可以用来对 tuples 应用 one-one transformation (一一变换)。

例如, 如果有一个 **stream of words** (单词流), 并且您想将其转换为 **stream of upper case words** (大写字母的流), 你可以定义一个 **mapping function** (映射函数) 如下,

```
public class UpperCase extends MapFunction {  
    @Override  
    public Values execute(TridentTuple input) {  
        return new Values(input.getString(0).toUpperCase());  
    }  
}
```

然后可以将 **mapping function** (映射函数) 应用于 **stream** 以产生 **stream of uppercase words** (大写字母的流) .

```
mystream.map(new UpperCase())
```

**flatMap** 类似于 **map**, 但具有将 **one-to-many transformation** (一对多变换) 应用于 **values of the stream** (流的值) 的效果, 然后将所得到的元素 **flattening** (平坦化) 为新的 **stream** .

例如, 如果有 **stream of sentences** (句子流), 并且您想将其转换成 **stream of words** (单词流), 你可以定义一个 **flatMap** 函数如下,

```
public class Split extends FlatMapFunction {  
    @Override  
    public Iterable<Values> execute(TridentTuple input) {  
        List<Values> valuesList = new ArrayList<>();  
        for (String word : input.getString(0).split(" ")) {  
            valuesList.add(new Values(word));  
        }  
        return valuesList;  
    }  
}
```

然后可以将 **flatMap** 函数应用于 **stream of sentences** (句子流) 以产生一个 **stream of words** (单词流) ,

```
mystream.flatMap(new Split())
```

当然这些操作可以被 **chained** (链接), 因此可以从如下的 **stream of sentences** (句子流) 中获得 **stream of uppercase words** (大写字母的流) ,

```
mystream.flatMap(new Split()).map(new UpperCase())
```

如果不将 **output fields** (输出字段) 作为 **parameter** (参数) 传递, 则 **map** 和 **flatMap** 会将 **input fields** (输入字段) 保留为 **output fields** (输出字段) .

如果要使用 `MapFunction` 或 `FlatMapFunction` 使用 `new output fields` (新的输出字段) 替换 `old fields` (旧字段), 您可以使用附加的 `Fields` 参数调用 `map/flatMap`, 如下所示,

```
mystream.map(new Uppercase(), new Fields("uppercased"))
```

`Output stream` (输出流) 只有一个 `output field` (输出字段) "uppercased", 而不管以前的流有什么输出字段. 同样的事情适用于 `flatMap`, 所以以下是有效的,

```
mystream.flatMap(new Split(), new Fields("word"))
```

## peek

`peek` 可用于在每个 `trident tuple` 流过 `stream` 时对其执行 `additional action` (附加操作). 这可能对于在流经 `pipeline` 中 `certain point` (某一点) 的元组来 `debugging` (调试) `tuples` 是有用的.

例如, 下面的代码将打印在将这些单词转换为 `groupBy` 之前将单词转换为大写的结果

```
java mystream.flatMap(new Split()).map(new Uppercase()) .peek(new Consumer() { @Override public void
```

## min and minBy

`min` 和 `minBy` operations (操作) 在 `trident stream` 中的 `a batch of tuples` (一批元组) 的每个 `partition` (分区) 上返回 `minimum value` (最小值).

假设 `trident stream` 包含字段 `["device-id", "count"]` 和 `partitions of tuples` (元组的以下分区)

```
Partition 0:
```

```
[123, 2]  
[113, 54]  
[23, 28]  
[237, 37]  
[12, 23]  
[62, 17]  
[98, 42]
```

```
Partition 1:
```

```
[64, 18]  
[72, 54]  
[2, 28]  
[742, 71]  
[98, 45]  
[62, 12]  
[19, 174]
```

Partition 2:

```
[27, 94]  
[82, 23]  
[9, 86]  
[53, 71]  
[74, 37]  
[51, 49]  
[37, 98]
```

`minBy` operation (操作) 可以应用在上面的 **stream of tuples** (元组流) 中, 如下所示, 这导致在每个 **partition** (分区) 中以最小值 `count` field (字段) 发出 **tuples**.

```
mystream.minBy(new Fields("count"))
```

上述代码在上述 **partitions** (分区) 上的结果是:

Partition 0:

```
[123, 2]
```

Partition 1:

```
[62, 12]
```

Partition 2:

```
[82, 23]
```

您可以在 **Stream** 上查看其他 `min` 和 `minBy` 操作

```
java public <T> Stream minBy(String inputFieldName, Comparator<T> comparator) public Stream min(Compar
```

下面的示例显示了如何使用这些 API 来使用 tuple 上的 **respective Comparators** (相应比较器) 来找到 **minimum** (最小值).

```
FixedBatchSpout spout = new FixedBatchSpout(allFields, 10, Vehicle.generateVehicles(20));  
  
TridentTopology topology = new TridentTopology();  
Stream vehiclesStream = topology.newStream("spout1", spout).  
    each(allFields, new Debug("##### vehicles"));  
  
Stream slowVehiclesStream =  
    vehiclesStream  
        .min(new SpeedComparator()) // Comparator w.r.t speed on received tuple.  
        .each(vehicleField, new Debug("##### slowest vehicle"));  
  
vehiclesStream  
    .minBy(Vehicle.FIELD_NAME, new EfficiencyComparator()) // Comparator w.r.t efficiency  
    .each(vehicleField, new Debug("##### least efficient vehicle"));
```

这些 API 的示例应用程序可以位于 [TridentMinMaxOfDevicesTopology](#) 和 [TridentMinMaxOfVehiclesTopology](#) .

## max and maxBy

`max` 和 `maxBy` operations (操作) 在 trident stream 中的一 batch of tuples (批元组) 的每个 partition (分区) 上返回 maximum (最大值) .

假设 trident stream 包含上述部分所述的字段 `["device-id", "count"]` .

`max` 和 `maxBy` operations (操作) 可以应用于上面的 stream of tuples (元组流) , 如下所示, 这导致每个分区的最大值为 `count` 字段的元组.

```
mystream.maxBy(new Fields("count"))
```

上述代码在上述 partitions (分区) 上的结果是:

```
Partition 0:  
[113, 54]
```

```
Partition 1:  
[19, 174]
```

```
Partition 2:  
[37, 98]
```

您可以在 Stream 上查看其他 `max` 和 `maxBy` 函数

```
public <T> Stream maxBy(String inputFieldName, Comparator<T> comparator)  
public Stream max(Comparator<TridentTuple> comparator)
```

下面的示例显示了如何使用这些 API 来使用元组上的 respective Comparators (相应比较器) 来找到 maximum (最大值) .

```
FixedBatchSpout spout = new FixedBatchSpout(allFields, 10, Vehicle.generateVehicles(20));  
  
TridentTopology topology = new TridentTopology();  
Stream vehiclesStream = topology.newStream("spout1", spout).  
    each(allFields, new Debug("##### vehicles"));  
  
vehiclesStream
```

```
.max(new SpeedComparator()) // Comparator w.r.t speed on received tuple.  
.each(vehicleField, new Debug("#### fastest vehicle"))  
.project(driverField)  
.each(driverField, new Debug("##### fastest driver"));  
  
vehiclesStream  
.maxBy(Vehicle.FIELD_NAME, new EfficiencyComparator()) // Comparator w.r.t efficiency  
.each(vehicleField, new Debug("#### most efficient vehicle"));
```

这些 API 的示例应用程序可以位于 [TridentMinMaxOfDevicesTopology](#) 和 [TridentMinMaxOfVehiclesTopology](#)

## Widnowing

Trident streams 可以 batches (批处理) 同一个 windowing (窗口) 的元组, 并将 aggregated result (聚合结果) 发送到下一个 operation (操作). 有 2 种支持的 windowing (窗口) 是基于 processing time (处理时间) 或 tuples count (元组数) : 1. Tumbling window 2. Sliding window

### Tumbling window

基于 processing time (处理时间) 或 count (计数), 元组在 single window (单个窗口) 中分组. 任何 tuple (元组) 只属于其中一个 windows (窗口).

```
/**  
 * Returns a stream of tuples which are aggregated results of a tumbling window with every {@code  
 */  
public Stream tumblingWindow(int windowCount, WindowsStoreFactory windowStoreFactory,  
                             Fields inputFields, Aggregator aggregator, Fields functionFields);  
  
/**  
 * Returns a stream of tuples which are aggregated results of a window that tumbles at duration  
 */  
public Stream tumblingWindow(BaseWindowedBolt.Duration windowDuration, WindowsStoreFactory windowStoreFactory,  
                             Fields inputFields, Aggregator aggregator, Fields functionFields);
```

### Sliding window

每个 sliding interval (滑动间隔), Tuples (元组) 被分组在 windows (窗口) 和 window slides 中. 元组可以属于多个 window (窗口).

```

/**
 * Returns a stream of tuples which are aggregated results of a sliding window with every {@code
 * and slides the window after {@code slideCount}.
 */
public Stream slidingWindow(int windowCount, int slideCount, WindowsStoreFactory windowStoreFac
    Fields inputFields, Aggregator aggregator, Fields functionFie

/**
 * Returns a stream of tuples which are aggregated results of a window which slides at duration
 * and completes a window at {@code windowDuration}
 */
public Stream slidingWindow(BaseWindowedBolt.Duration windowDuration, BaseWindowedBolt.Duration
    WindowsStoreFactory windowStoreFactory, Fields inputFields, Agg

```

tumbling 和 sliding windows 的示例可以在 [这里](#) 被找到.

## 通用 windowing API

以下是通用的 windowing API, 它为任何支持的 windowing 配置提供了 `WindowConfig` .

```

public Stream window(WindowConfig windowConfig, WindowsStoreFactory windowStoreFactory, Fields
    Aggregator aggregator, Fields functionFields)

```

`windowConfig` 可以是下面的任何一个. - `SlidingCountWindow.of(int windowCount, int slidingCount)` -  
`SlidingDurationWindow.of(BaseWindowedBolt.Duration windowDuration, BaseWindowedBolt.Duration slidingDura`  
- `TumblingCountWindow.of(int windowLength)` -  
`TumblingDurationWindow.of(BaseWindowedBolt.Duration windowLength)`

Trident windowing APIs 需要 `WindowsStoreFactory` 来存储接收的 tuples 和 aggregated values (聚合值) . 目前, HBase 的基本实现由 `HBaseWindowsStoreFactory` 提供. 可以进一步扩展以解决各自的用途. 使用 `HBaseWindowStoreFactory` 进行 windowing 的例子可以在下面看到.

```

// window-state table should already be created with cf:tuples column
HBaseWindowsStoreFactory windowStoreFactory = new HBaseWindowsStoreFactory(new HashMap<String,
FixedBatchSpout spout = new FixedBatchSpout(new Fields("sentence"), 3, new Values("the cow jump
    new Values("the man went to the store and bought some candy"), new Values("four score a
    new Values("how many apples can you eat"), new Values("to be or not to be the person"))
spout.setCycle(true);

TridentTopology topology = new TridentTopology();

Stream stream = topology.newStream("spout1", spout).parallelismHint(16).each(new Fields("senten
    new Split(), new Fields("word"))
    .window(TumblingCountWindow.of(1000), windowStoreFactory, new Fields("word"), new Count
    .peek(new Consumer() {

```

```
    @Override
    public void accept(TridentTuple input) {
        LOG.info("Received tuple: {}", input);
    }
);

StormTopology stormTopology = topology.build();
```

可以在 [这里](#) 中找到本节中所有上述 API 的详细说明.

## 示例应用程序

这些 API 的示例应用程序位于 [TridentHBaseWindowingStoreTopology](#) 和 [TridentWindowingInmemoryStoreTopology](#)

### partitionAggregate

partitionAggregate 在每个 batch of tuples (批量元组) partition 上执行一个 function 操作 (实际上是聚合操作), 但它又不同于上面的 functions 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("b"), new Sum(), new Fields("sum"))
```

假设 input stream 包括字段 ["a", "b"] , 并有下面的 partitions of tuples (元组 partitions) :

Partition 0:

```
["a", 1]
["b", 2]
```

Partition 1:

```
["a", 3]
["c", 8]
```

Partition 2:

```
["e", 1]
["d", 9]
["d", 10]
```

则这段代码的 output stream 包含如下 tuple , 且只有一个 "sum" 的字段:

Partition 0:

```
[3]
```

Partition 1:

```
[11]
```

Partition 2:

上面代码中的 new Sum() 实际上是一个 aggregator (聚合器), 定义一个聚合器有三种不同的接口:CombinerAggregator, ReducerAggregator 和 Aggregator .

下面是 CombinerAggregator 接口:

```
public interface CombinerAggregator<T> 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.下面是聚合器 Count 的实现:

```
public class Count implements CombinerAggregator<Long> {
    public Long init(TridentTuple tuple) {
        return 1L;
    }

    public Long combine(Long val1, Long val2) {
        return val1 + val2;
    }

    public Long zero() {
        return 0L;
    }
}
```

当使用 aggregate() 方法代替 partitionAggregate() 方法时, 就能看到 CombinerAggregation 带来的好处.这种情况下, Trident 会自动优化计算:先做局部聚合操作, 然后再通过网络传输 tuple 进行全局聚合.

ReducerAggregator 接口如下:

```
public interface ReducerAggregator<T> 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 .下面示例了如何将 Count 定义为 ReducerAggregator:

```

public class Count implements ReducerAggregator<Long> {
    public Long init() {
        return 0L;
    }

    public Long reduce(Long curr, TridentTuple tuple) {
        return curr + 1;
    }
}

```

ReducerAggregator 也可以与 persistentAggregate 一起使用, 稍后你会看到的.

用于 performing aggregations (执行聚合) 的最通用的接口是 Aggregator , 如下所示:

```

public interface Aggregator<T> 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 update the state (更新状态) (第一点中 init() 方法的返回值) 并 optionally emit tuples (可选地发出元组) .
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.

注:理解 batch, partition 之间的区别将会更好的理解上面的几个方法.

下面的代码将 Count 作为 Aggregator 实现:

```

public class CountAgg extends BaseAggregator<CountState> {
    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));  
}  
}
```

有时需要同时执行 **multiple aggregators** (多个聚合) 操作, 这个可以使用 **chaining** (链式) 操作完成:

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

这段代码将会对每个 **partition** 执行 **Count** 和 **Sum aggregators** (聚合器), 并输出一个**tuple** 字段 **["count", "sum"]**.

## stateQuery and partitionPersist

**stateQuery** 和 **partitionPersist** 分别 **query** (查询) 和 **update** (更新) **sources of state** (状态源). 您可以在 [Trident state doc](#) 上阅读有关如何使用它们.

## 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). 如果你有一个包含字段 **["a", "b", "c", "d"]** 的 **stream**, 执行下面代码:

```
mystream.project(new Fields("b", "d"))
```

则 **output stream** 将仅包含 **["b", "d"]** 字段.

## Repartitioning operations

**Repartitioning operations** (重新分区操作) 运行一个函数来 **change how the tuples are partitioned across tasks** (更改元组在任务之间的分区). **number of partitions** (分区的数量) 也可以由于 **repartitioning** (重新分区) 而改变 (例如, 如果并行提示在 **repartitioning** (重新分配) 后更大). **Repartitioning** (重新分区) 需要 **network transfer** (网络传输). 以下是 **repartitioning functions** (重新分区功能) :

1. **shuffle**: 随机将 **tuple** 均匀地分发到目标 **partition** 里.
2. **broadcast**: 每个 **tuple** 被复制到所有的目标 **partition** 里, 在 **DRPC** 中有用 — 你可以在每个 **partition** 上使用 **stateQuery** .

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 ) .
4. `global`: 所有的 tuple 都被发送到同一个 partition .
5. `batchGlobal`: 确保同一个 batch 中的 tuple 被发送到相同的 partition 中.
6. `partition`: 此方法采用实现 `org.apache.storm.grouping.CustomStreamGrouping` 的自定义分区函数.

## Aggregation operations

Trident 中有 `aggregate()` 和 `persistentAggregate()` 方法对流进行聚合操作. `aggregate()` 在每个 batch 上独立的执行, `persistemAggregate()` 对所有 batch 中的所有 tuple 进行聚合, 并将结果存入 state 源中.

`aggregate()` 对 Stream 做全局聚合, 当使用 `ReduceAggregator` 或者 `Aggregator` 聚合器时, 流先被重新划分成一个大分区(仅有一个 partition ), 然后对这个 partition 做聚合操作;另外, 当使用 `CombinerAggregator` 时, Trident 首先对每个 partition 局部聚合, 然后将所有这些 partition 重新划分到一个 partition 中, 完成全局聚合.相比而言, `CombinerAggregator` 更高效, 推荐使用.

下面的例子使用 `aggregate()` 对一个 batch 操作得到一个全局的 count:

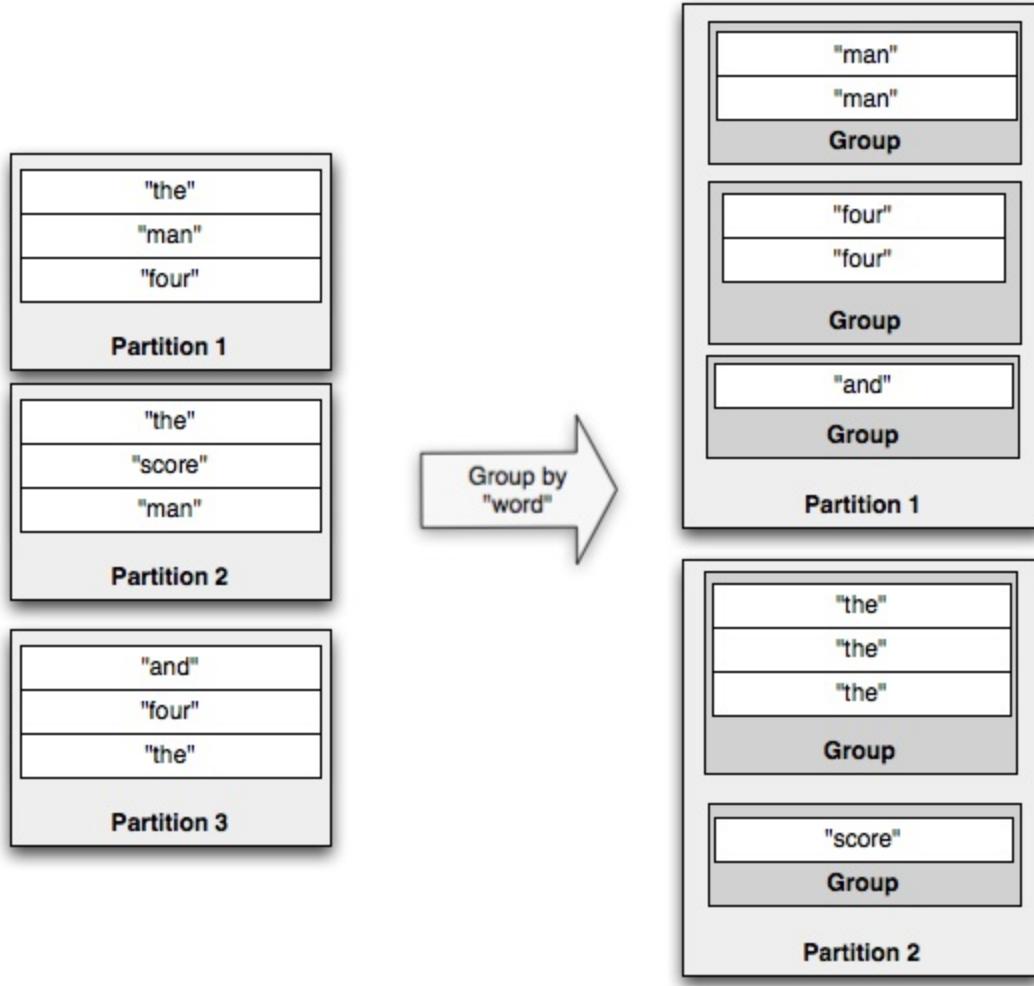
```
mystream.aggregate(new Count(), new Fields("count"))
```

同在 `partitionAggregate` 中一样, `aggregate` 中的聚合器也可以使用链式用法.但是, 如果你将一个 `CombinerAggregator` 链到一个非 `CombinerAggregator` 后面, Trident 就不能做局部聚合优化.

关于 `persistentAggregate` 的用法请参见 [Trident state doc](#) 一文.

## Operations on grouped streams

`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:



如果你在一个 grouped stream 上做聚合操作, 聚合操作将会在每个 group (分组) 内进行, 而不是整个 batch 上. GroupStream 类中也有 persistentAggregate 方法, 该方法聚合的结果将会存储在一个 key 值为分组字段(即 groupBy 中指定的字段)的 MapState 中, 这些还是在 [Trident state doc](#) 一文中讲解.

和普通的 stream 一样, groupstream 上的聚合操作也可以使用 chained (链式语法) .

## Merges and joins

最后一部分 API 内容是关于将几个 stream 汇总到一起, 最简单的汇总方法是将他们合并成一个 stream , 这个可以通过 TridentTopology 中的 merge 方法完成, 就像这样:

```
topology.merge(stream1, stream2, stream3);
```

Trident 将把新的 merged stream 的 output fields 命名为第一个 stream 的 output fields (输出字段) .

另一种 combine streams (汇总方法) 是使用 join (连接, 类似于 sql 中的连接操作, 需要有限的输入) .所以, 它们对于 infinite streams (无限流) 是没有意义的. Joins in Trident 仅

适用于从 **spout** 发出的每个 **small batch** 中.

以下是包含字段 **["key", "val1", "val2"]** 的 **stream** 和包含 **["x", "val1"]** 的另一个 **stream** 之间的 **join** 示例:

```
topology.join(stream1, new Fields("key"), stream2, new Fields("x"), new Fields("key", "a", "b", "c")
```

使用 **"key"** 和 **"x"** 作为每个相应流的连接字段将 **stream1** 和 **stream2** **join** 在一起.然后, Trident 要求命名 **new stream** 的所有 **output fields** , 因为 **input streams** 可能具有 **overlapping field names** (重叠的字段名称) .从 **join** 发出的 **tuples** 将包含:

1. **list of join fields** (连接字段列表) .在这种情况下, **"key"** 对应于 **stream1** 的 **"key"** , **stream2** 对应于 **"x"** .
2. 接下来, 按照 **streams** 如何传递到 **join** 方法的顺序, 所有流中的所有 **non-join fields** (非连接字段) 的列表.在这种情况下, **"a"** 和 **"b"** 对应于来自 **stream1** 的 **"val1"** 和 **"val2"** , **"c"** 对应于来自 **stream2** 的 **"val1"** .

当来自不同 **spouts** 的 **stream** 之间发生 **join** 时, 这些 **spouts** 将与它们如何 **emit batches** (发出批次) 同步.也就是说, 一批处理将包括 **tuples from each spout** (每个 **spout** 的元组) .

你可能会想知道 - 你如何做一些像 **"windowed join"** 这样的事情, 其中从 **join** 的一边的 **tuples** 连接 **join** 另一边的最后个小时的 **tuples** .

为此, 您将使用 **partitionPersist** 和 **stateQuery** .**join** 一端的元组的最后一小时将被存储并在 **source of state** (状态源) 中旋转, 并由 **join** 字段键入.然后 **stateQuery** 将通过连接字段进行查找以执行 **"join"**.

# Trident State

Trident 拥有一流的 abstractions（抽象）用于 reading from（读取）和 writing to（写入）stateful sources（状态源）. state（状态）可以是 internal to the topology（拓扑内部），例如保存在内存中并由 HDFS 支持 - 或者 externally stored（外部存储）在像 Memcached 或者 Cassandra 这样的数据库中. Trident API 在任何一种情况下都没有区别.

Trident 以 fault-tolerant（容错方式）来管理 state（状态），以便在 retries（重试）和 failures（失败）时 state updates（状态更新）是 idempotent（幂等）的. 这可以让您理解 Trident topologies（Trident 拓扑结构），就好像每个消息都被 exactly-once（精确处理一次）.

在进行 state updates（状态更新）时可能会有各种级别的 fault-tolerance（容错能力）. 在得到这些之前，我们来看一个例子来说明实现 exactly-once semantics（完全一次性语义）所必需的技巧. 假设您正在对 stream（流）进行 count aggregation（计数聚合），并希望将 running count（运行的计数）存储在数据库中. 现在假设您在数据库中存储一个 single value representing the count（表示计数的值），并且每次处理 new tuple（新的元组）时，都会增加 count（计数）.

发生故障时，将 replayed tuples（元组）. 这会导致在执行 state updates（状态更新）时出现问题（或任何带有副作用的东西） - 您不知道如果您曾经成功根据此 tuple（元组）更新了 state（状态）. 也许你从来没有处理过 tuple（元组），在这种情况下你应该增加 count（计数）. 也许你已经处理了 tuple，并成功地增加了 count（计数），但是 tuple（元组）在另一个步骤中处理失败. 在这种情况下，您不应该增加 count（计数）. 或者也许你看到了 tuple，但更新数据库时出错. 在这种情况下，您应该 update the database（更新数据库）.

通过将 count 存储在数据库中，您不知道这个 tuple 是否被处理过. 所以你需要更多的信息才能做出正确的决定. Trident 提供以下 semantics（语义），足以实现 exactly-once（完全一次性）处理语义：

1. Tuples（元组）被 small batches（小批）处理（参见 [the tutorial](#)）
2. Each batch of tuples（每批元组）给出一个唯一的id，称为 "transaction id" (txid). 如果 batch（批次）被 replayed，则给出完全相同的 txid .
3. State updates（状态更新）跟随 batches（批次）的顺序. 也就是说，在 batch 2 的状态更新成功之前，batch 3 的状态更新将不会被应用.

使用这些 primitives (原语), 您的 State implementation (State 实现) 可以检测该 batch (批次) 的 tuples (元组) 是否已被处理, 并采取适当的操作以一致的方式更新状态. 您所采取的操作取决于您的 input spouts (输入端口) 提供的关于每 batch 中的内容的 exact semantics (确切语义). 有关 fault-tolerance (容错) 的 spouts 可能性有三种:"non-transactional (非事务性)", "transactional (事务)" 和 "opaque transactional (不透明事务)". 同样, 在 fault-tolerance (容错) 方面有三种可能 state : "non-transactional (非事务性)", "transactional (事务)" 和 "opaque transactional (不透明事务)". 我们来看看每个 spout 类型, 看看你可以实现什么样的容错.

## Transactional spouts

请记住, Trident 将 tuples (元组) 作为 small batches (小批量) 进行处理, 每个批处理都被赋予一个唯一的 transaction id (事务 ID). spouts 的属性根据他们可以提供的每 batch (批次) 中的内容的 guarantees (保证) 而有所不同. transactional spout 具有以下属性:

1. 给定 txid 的 Batches (批次) 总是相同的. txid 的批次的 Replays 将与该 txid 的第一次发出批次时的一组元组完全相同.
2. batches of tuples (批量的元组) 之间没有 overlap (重叠) ( tuples (元组) 在一个批次或另一个批次中, 而不是在多个批次中) .
3. 每个元组都是批量的 (没有元组被跳过) .

这是一个理解起来非常简单的类型的 spout , stream 被分为 fixed batches (固定批次), 从不改变. storm-contrib 具有 [一个 transactional spout 的实现](#) 针对于 Kafka .

你可能会想 - 为什么你不总是使用一个 transactional spout ? 它们简单易懂. 你不能使用它的一个原因是它们不一定非常 fault-tolerant (容错). 例如, TransactionalTridentKafkaSpout 的工作原理是 txid 的批处理将包含来自所有 Kafka partitions 的元组. 一旦批次被发出, 在未来的任何时候批次被重新发出, 必须发出完全相同的元组集合才能满足 transactional spouts 的语义. 现在假设一个批处理从 TransactionalTridentKafkaSpout 发出, batch 无法处理, 同时一个 Kafka 节点宕机. 您现在无法像以前一样 replaying 同一批次 (因为节点关闭, topic 的某些 partitions 不可用), 并且处理将 halt (停止) .

这就是为什么存在 "opaque transactional (不透明事务)" spout - 它们容许这样的错误, 丢失 source nodes (源节点), 同时仍允许您实现 exactly-once (一次) 处理语义. 我们将在

下一节中介绍这些 **spouts** .

(一方面注意 - 一旦 Kafka 支持 replication (备份) , 就有可能拥有对节点故障容错的 **transactional spouts** , 但该功能尚不存在. )

在我们介绍 "opaque transactional (不透明事务) " **spouts** 之前, 我们来看看如何设计一个具有 **exactly-once semantics** (完全一致的语义的) **transactional spouts** 的 State implementation (状态实现) . 这种状态称为 "transactional state (事务状态) " , 并且利用了任何给定的 txid 始终与完全相同的元组集合相关联的事实.

假设您的 topology 计算 word count , 并且您想将 word counts 存储在 key/value database 中. key 将是这个 word , value 将包含 count . 您已经看到, 仅将 count 存储为 value 不足以知道是否已经处理了一批元组. 相反, 您可以做的是将 transaction id 存储在数据库中的 count 作为 atomic value (原子值) . 然后, 当更新 count 时, 可以将数据库中的 transaction id 与当前批处理的 transaction id 进行比较. 如果它们相同, 则跳过该更新 - 由于强大的排序, 您可以确定数据库中的 value 包含当前 batch . 如果它们不同, 你会增加 count . 此逻辑工作原理是因为 txid 的批次不会更改, Trident 可确保 state updates 跟随 batches 的顺序.

考虑这个为什么它起作用的例子. 假设您正在处理由以下批次元组组成的 txid 3 :

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

假设数据库当前持有以下 key/value 对:

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

与 "man" 相关联的 txid 为 txid 1 . 由于当前的 txid 为 3 , 因此您可以肯定地知道这批元组在该 count 中未被显示. 所以你可以继续递增 count 2 并更新 txid . 另一方面, "dog" 的 txid 与当前的 txid 相同. 所以你确定当前批次的增量已经在数据库中被显示为 "dog" key . 所以你可以跳过更新. 完成更新后, 数据库如下所示:

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

现在我们来看看 opaque transactional spouts (不透明事务 spouts) , 以及如何设计这种

spout 的 states.

## Opaque transactional spouts

如前所述, opaque transactional spout 不能保证 txid 的元组的批次保持不变. opaque transactional spout 具有以下属性:

1. 每个元组都被 **exactly one batch** (正确的一个批次中) *successfully* 处理. 但是, 一个元组可能无法在一个批处理中处理, 然后在后续批处理中成功处理.

[OpaqueTridentKafkaSpout](#) 是一个具有此属性并且是对丢失 Kafka 节点有容错性的 spout . 无论何时 OpaqueTridentKafkaSpout emit a batch (发出批次), 它将从最后一批完成发出的位置开始发出元组. 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 spouts , 如果数据库中的 transaction id 与当前批处理的 transaction id 相同, 则不再可能使用 **trick of skipping state updates** (跳过状态更新的技巧) . 这是因为在 state updates (状态更新) 之间批处理可能已更改.

你可以做的是在数据库中存储更多的 state . 而不是在数据库中存储 value 和 transaction id , 而是将 value , transaction id 和上一个 value 存储在数据库中. 我们再次使用在数据库中存储 count 的示例. 假设您的批次的部分计数是 "2" , 现在是应用 state update (状态更新) 的时间. 假设数据库中的 value 如下所示:

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

假设你当前的 txid 是 3 , 与数据库不同. 在这种情况下, 您将 "prevValue" 设置为 "value" , 通过 partial count 增加 "value" , 并更新 txid . 新的数据库值将如下所示:

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

现在假设你当前的 txid 是 2 , 等于数据库中的内容. 现在, 您知道数据库中的 "value" 包含来自当前 txid 的上一批次的更新, 但该批次可能已经不同, 因此您必须忽略它. 在这种情况下, 您的 partial count 将增加 "prevValue" , 以计算新的 "value" . 然后将数据库中的 value 设置

为:

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

这是因为 Trident 提供的批次的 **strong ordering** (强大顺序) . 一旦 Trident 移动到新的批次进行状态更新, 它将永远不会返回到上一批. 而且由于 **opaque transactional spouts** 保证批次之间不 **overlap** (重叠) - 每个元组都被一个批次成功处理 - 您可以根据先前的 **value** 安全地进行更新.

## Non-transactional spouts

Non-transactional spouts 不对每批中的内容提供任何保证. 所以它可能是 **at-most-once** (最多一次) 的处理, 在这种情况下, 元组不会在失败的批次后重试. 或者它可能 **at-least-once** (至少处理一次), 其中可以通过多个批次成功处理元组. 没有办法为这种 **spout** 实现 **exactly-once semantics** (完全一次性语义) .

## spout 和 state type 的汇总

此图显示了 spouts / states 的哪些组合可以实现一次消息传递语义:

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

Opaque transactional states 具有最强的 **fault-tolerance** (容错能力), 但这需要以 txid 和两个 values 存储在数据库中为代价. Transactional states 在数据库中需要较少的 state , 但仅适用于 transactional spouts . 最后, non-transactional states 在数据库中需要最少的 state , 但不能实现 **exactly-once semantics** (一次性语义) .

您选择的 **state** 和 **spout types** 是容错和存储成本之间的折中, 最终您的应用程序要求将决定哪种组合适合您.

## State APIs

您已经看到了完成 **exactly-once semantics** (完全一次语义) 所需要的复杂性. **Trident** 的好处是它将所有容错逻辑内部化 - 作为一个用户, 您不必处理比较 txids , 在数据库中存储多个值或类似的内容. 你可以这样编写代码:

```
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);
```

管理 **opaque transactional state logic** 所需的所有 **logic** 都内在于 **MemcachedState.opaque** 调用. 此外, 自动批量更新以最小化到数据库的 **roundtrips** (往返行程) .

基本 **State interface** (状态接口) 只有两种方法:

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

当状态更新开始时, 您被告知, 当状态更新结束时, 在每种情况下都被给予了 **txid** . **Trident** 对于你的 **state** 如何工作, 什么样的方法有更新, 以及从中读取什么样的方法呢?

假设您有一个 **home-grown database** (本地生成的数据库), 其中包含用户位置信息, 并且您希望能够从 **Trident** 访问它. 您的 **State implementation** 将具有获取和设置用户信息的方法:

```
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, 您可以向 Trident 提供一个 StateFactory , 它可以在 Trident tasks 中创建 State 对象的实例. 您的 LocationDB 的 StateFactory 可能看起来像这样:

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

Trident 提供 QueryFunction interface , 用于编写查询 source of state (状态源) 的 Trident 操作, 以及 StateUpdater 接口, 用于编写更新 source of state (状态源) 的 Trident 操作. 例如, 我们来写一个操作 "QueryLocation" , 它查询 LocationDB 的用户位置. 我们先来看看如何在 topology 中使用它. 假设这种 topology 消耗了 userids 的 input stream:

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

现在我们来看看 QueryLocation 的实现如何:

```
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 的两个步骤执行. 首先, Trident 将一批读取合并在一起, 并将它们传递给 batchRetrieve . 在这种情况下, batchRetrieve 将接收 multiple user ids (多个用户 ID) . batchRetrieve 预期返回与输入元组列表大小相同的结果列表. 结果列表的第一个元素对应于第一个输入元组的结果, 第二个是第二个输入元组的结果, 依此类推.

你可以看到, 这个代码没有利用 Trident 的批处理, 因为它只是一次查询一个 LocationDB . 所

以写一个更好的方法来编写 LocationDB 就是这样的:

```
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  
    }  
}
```

然后，您可以像这样编写 QueryLocation 函数:

```
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));  
    }  
}
```

通过减少到数据库的 roundtrips (往返行程)，此代码将更加高效.

要 update state，可以使用 StateUpdater interface . 这是一个 StateUpdater，它使用新的位置信息来更新 LocationDB :

```
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);  
}
```

以下是在 Trident topology 中使用此操作的方法:

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

partitionPersist 操作更新 source of state (状态源) . StateUpdater 收到该 State 和一批具有该 State 更新的元组. 该代码只是从输入元组中获取用户名和位置, 并将批量集合放入 States .

partitionPersist 返回表示由 Trident topology 更新的位置数据块的 TridentState 对象. 然后, 您可以在 topology 中的其他地方的 stateQuery 操作中使用此 state .

您还可以看到 StateUpdaters 被赋予了 TridentCollector . 发送到这个 collector 的元组转到 "new values stream" . 在这种情况下, 没有什么有趣的可以发送到该 stream , 但是如果您在数据库中进行更新 counts , 则可以将更新的 counts 发送到该 stream . 然后, 您可以通过 TridentState#newValuesStream 方法访问 new values stream 以进一步处理.

## persistentAggregate

Trident 有另外一种更新 State 的方法叫做 persistentAggregate . 你在之前的 streaming 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 aggregator (Trident 聚合器) 来更新 State . 在这个例子当中, 因为这是一个 grouped stream (分组流) , 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 non-grouped streams 上面进行 aggregations (聚合) 的话, Trident 会期待你的 State 对象实现 "Snapshottable" 接口:

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

[MemoryMapState](#) 和 [MemcachedState](#) 分别实现了上面的 2 个接口.

## Implementing 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. 你只需要将一个知道如何执行相应 key/values 的 multiGet 和 multiPuts 的 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](#) 会用 [OpaqueValue](#) 的 value 来调用 multiPut 方法, [TransactionalMap](#) 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 (缓存) map key/vals .

最后, Trident 提供了 [SnapshottableMap](#) 类, 通过将 global aggregations (全局聚合) 存储到 fixed key (固定密钥) 中将一个 MapState 转换成一个 Snapshottable 对象.

大家可以看看 [MemcachedState](#) 的实现, 从而学习一下怎样将这些工具组合在一起形成一个高性能的 MapState 实现. MemcachedState 是允许你选择使用 opaque transactional , transactional , 还是 non-transactional 语义的.

# Trident Spouts

## Trident spouts

像在 vanilla Storm API 中, spouts 是 Trident topology 中的 source of streams (streams 的来源) . 在 vanilla Storm spouts 之上, Trident 为更复杂的 spouts 提供了额外的 API .

您的 how you source your data streams (数据流的来源) 和 how you update state (e.g. databases) based on those data streams (基于这些数据流更新状态 (例如数据库)) 之间存在着一个不可分割的联系. 请参阅 [Trident state doc](#) , 以了解此信息 - 了解此联系对于了解可用的 spout 选项是至关重要的.

Regular Storm spouts 将是 Trident topology 中的 non-transactional spouts . 要使用 regular Storm IRichSpout, 请在 TridentTopology 中创建如下 stream :

```
TridentTopology topology = new TridentTopology();
topology.newStream("myspoutid", new MyRichSpout());
```

Trident topology 中的所有 spouts 都需要为该流提供 unique identifier (唯一的标识符) - 该 identifier 在集群上运行的所有 topologies 中必须是唯一的. Trident 将使用此 identifier 来存储 Zookeeper 中 spout 消耗的 metadata (元数据), 包括 txid 和与 spout 相关联的任何 metadata (元数据) .

您可以通过以下配置选项配置 spout metadata 的 Zookeeper 存储:

1. `transactional.zookeeper.servers`: Zookeeper hostnames 列表
2. `transactional.zookeeper.port`: Zookeeper 集群的端口
3. `transactional.zookeeper.root`: Zookeeper 中存储 Metadata 的根目录. Metadata 将存储在路径 /

## Pipelining

默认情况下, Trident 一次处理 single batch (一个批次), 等待批次成功或失败, 然后再尝试处理 another batch (另一批次). 您可以通过 pipelining the batches 获得 significantly higher throughput (明显更高的吞吐量) 并降低每个 batch (批处理) 的延迟. 您可以使用 "topology.max.spout.pending" 属性同时处理要处理的 maximum amount of batches (最大批处理量) .

即使在同时处理 **multiple batches** (多个批次) 的同时, Trident 也会在 **topology** 中将任何 **state updates** 按照 **batches** 的顺序排序. 例如, 假设您正在对数据库进行全局计数聚合. 这个想法是, 当您更新数据库中 **batch 1** 的计数时, 您仍然可以计算 **batch 2** 到 **10** 的部分计数. Trident 将不会移动到 **batch 2** 的 **state updates** (状态更新), 直到状态更新为 **batch 1** 已成功. 这对于实现 **exactly-once** (一次且仅一次) 处理语义非常重要, 如 [Trident state doc](#) 中的大纲.

## Trident spout types

以下是可用的以下 spout API:

1. [ITridentSpout](#): 最通用的 API, 可以支持 **transactional** 或 **opaque transactional semantics** (语义). 一般来说, 您将直接使用此 API 的一种 **partitioned** (分区) 风格, 而不是使用此 API .
2. [IBatchSpout](#): 一次发送 **batches of tuples** 的 **non-transactional** spout .
3. [IPartitionedTridentSpout](#): 从 **partitioned data source** (分区数据源) 读取的 **transactional** spout (像 Kafka 服务器集群)
4. [IOpaquePartitionedTridentSpout](#): 一个从 **partitioned data source** (分区数据源) 读取的 **opaque transactional** spout .

而且, 像本教程开头所提到的, 你也可以使用常规的 [IRichSpout](#) .

# Trident RAS API

## Trident RAS API

Trident RAS ( Resource Aware Scheduler (资源感知调度程序) ) API 提供了一种机制, 允许用户指定 Trident topology 的 resource consumption (资源消耗) . API 看起来与基本的 RAS API 完全相同, 只是在 Trident Streams 上调用, 而不是 Bolts 和 Spouts .

为了避免文档中的 duplication (重复) 和 inconsistency (不一致) , resource setting (资源设置) 的目的和效果在这里不再描述, 而是在 [Resource Aware Scheduler Overview](#) 中可以找到,

## Use

首先, 例如:

```
TridentTopology topo = new TridentTopology();
topo.setResourceDefaults(new DefaultResourceDeclarer());
                           .setMemoryLoad(128)
                           .setCPULoad(20));

TridentState wordCounts =
    topology
        .newStream("words", feeder)
        .parallelismHint(5)
        .setCPULoad(20)
        .setMemoryLoad(512, 256)
        .each( new Fields("sentence"), new Split(), new Fields("word"))
        .setCPULoad(10)
        .setMemoryLoad(512)
        .each(new Fields("word"), new BangAdder(), new Fields("word!"))
        .parallelismHint(10)
        .setCPULoad(50)
        .setMemoryLoad(1024)
        .each(new Fields("word!"), new QMarkAdder(), new Fields("word!?"))
        .groupBy(new Fields("word!"))
        .persistentAggregate(new MemoryMapState.Factory(), new Count(), new Fields("count"))
        .setCPULoad(100)
        .setMemoryLoad(2048);
```

可以为每个操作设置 Resources (资源) (除了 grouping (分组), shuffling (混洗), partitioning (分区)) . 将 Trident combined 成 single Bolts 的操作将将其资源相加.

每个 Bolt 都被给予 至少 默认资源, 无论用户设置如何.

在上述情况下, 我们最终得到

- 一个 spout 和 spout coordinator (spout 协调器), 每个 CPU 负载为 20% , 内存负载为

512MiB , off-heap (堆栈) 为 256MiB .

- 组合的 `split` 和 `BangAdder` 以及 `QMarkAdder` 的 on-heap (堆栈) 中, 具有80% cpu 负载 (10%+ 50%+ 20%) 和 1664MiB (1024 + 512 + 128) 的内存负载的 bolt , 使用 `DefaultResourceDeclarer` 中包含的默认资源
- 具有 100% cpu 加载和 2048MiB 堆内存负载的 bolt , 默认值为 off-heap (非堆)

任何 `operation` (操作) 后都可以调用 `Resource declarations` (资源声明) . 没有明确资源的操作将获得默认值. 如果您选择仅为某些操作设置资源, 则必须声明默认值, 否则拓扑提交将失败. 资源声明具有与并行提示相同的 `boundaries` . 他们不会进行任何分组, 洗牌或任何其他类型的重新分配. 每个操作都会声明资源, 但在边界内组合.

# Storm SQL

# Storm SQL 集成

Storm SQL 使用用户在 Storm 中的流数据上运行 SQL 查询. SQL 接口不仅可以加快流数据分析的开发周期, 同时还创造了一个机遇, 统一如 [Apache Hive](#) 和实时流数据分析之类的批量数据处理.

在很高的级别, StromSQL 把 SQL 编译为 [Trident](#) 拓扑并在 Strom 集群中执行. 本文档提供了作为一个末端用户如何使用 StromSQL 的相关信息. 对于想更深入了解 StromSQL 的设计和实现的朋友请参考[这个](#) 页面.

Storm SQL 是一个 [试验性](#) 的功能, 因此其内部逻辑和支持的特性可能在将来会有变化. 但是小的改动不会影响用户体验. 在引入 [UX](#) 更改时, 我们会提醒和通知用户.

## 使用

运行 `storm sql` 命令把 SQL 语句编译为 [Trident topology](#), 并且提交到 Storm 集群.

```
$ bin/storm sql <sql-file> <topo-name>
```

`sql-file` 文件中包含需要被执行的 SQL 语句的列表, `topo-name` 是 `topology` 的名称.

当用户把 `topo-name` 设置为 `--explain` 的时候, StromSQL 激活 `explain mode` 以显示查询计划而不是提交拓扑. 详细的解释请参见 [显示查询计划\(explain mode\)](#) 一节.

## 支持的特性

当前版本支持以下特性:

- 读出和流入外部数据源
- 过滤 `tuples`
- 投影
- 用户自定义函数 (标量)

特意不支持聚合和连接. 当 Storm SQL 要支持本地 `Streaming SQL` 时, 将会介绍这些特性.

## 指定外部数据源

在 StromSQL 中, 数据表现为外部表. 用户可以使用语句 `CREATE EXTERNAL TABLE` 指定数据源.

`CREATE EXTERNAL TABLE` 语法与 [Hive Data Definition Language](#) 中的非常接近.

```
CREATE EXTERNAL TABLE table_name field_list
```

```
[ STORED AS  
  INPUTFORMAT input_format_classname  
  OUTPUTFORMAT output_format_classname  
]  
LOCATION location  
[ PARALLELISM parallelism ]  
[ TBLPROPERTIES tbl_properties ]  
[ AS select_stmt ]
```

各种属性的详细解释参考 [Hive Data Definition Language](#).

`PARALLELISM` 是 StormSQL 特有的关键词, 用于描述输入数据源的并行度. 等同于为 Trident Spout 设置并行度.

默认值是 1, 这个选项对于输出数据源没有任何影响. (如果需要的话, 以后可能会改变. 正常情况下应当避免重新分区).

例如, 下面的语句指定了一个 Kafka Spout 和 sink:

```
CREATE EXTERNAL TABLE FOO (ID INT PRIMARY KEY) LOCATION 'kafka://localhost:2181/brokers?topic=test'
```

## 植入外部数据源

用户通过实现 `ISqlTridentDataSource` 接口并且利用 Java 的 service loader 机制注册他们, 以植入外部数据源. 外部数据源将根据表的 URI 模式进行选择. 更多细节请参考 [storm-sql-kafka](#).

## 指定 User Defined Function (UDF)

用户可以使用 `CREATE FUNCTION` 语句来定义 user defined function (标量 或者 聚合). 例如, 下面的语句使用 `org.apache.storm.sql.TestTools$MyPlus` 类定义了一个名为 `MYPLUS` 的函数.

```
CREATE FUNCTION MYPLUS AS 'org.apache.storm.sql.TestTools$MyPlus'
```

Storm SQL 通过检查用了什么方法来决定这个函数作为一个 标量 还是 聚合. 如果类中定义了 `evaluate` 方法, Storm SQL 将这个函数作为 `scalar`.

标量函数类的示例:

```
public class MyPlus {  
    public static Integer evaluate(Integer x, Integer y) {  
        return x + y;  
    }  
}
```

## 例子: 过滤 Kafka 流

假设有一个 Kafka stream 代表订单交易. 每个 stream 中的消息包含订单的 id, 产品的单价, 产品数量. 目标是过滤重要交易的订单(译注:总价格大于50的订单), 并将这些订单插入到另一个 Kafka stream 用于进一步分析.

用户可以在 SQL 文件中指定下列 SQL 语句:

```
CREATE EXTERNAL TABLE ORDERS (ID INT PRIMARY KEY, UNIT_PRICE INT, QUANTITY INT) LOCATION 'kafka://127.0.0.1:9092/topics/orders'
CREATE EXTERNAL TABLE LARGE_ORDERS (ID INT PRIMARY KEY, TOTAL INT) LOCATION 'kafka://localhost:2181/topics/large_orders'
INSERT INTO LARGE_ORDERS SELECT ID, UNIT_PRICE * QUANTITY AS TOTAL FROM ORDERS WHERE UNIT_PRICE * QUANTITY > 50
```

第一个语句定义一个表 `ORDER` 代表输入流. `LOCATION` 从句指定 ZkHost (`localhost:2181`), broker 的路径(`/brokers`), 和 topic名称(`orders`).

类似的, 第二个语句指定了表 `LARGE_ORDERS` 代表一个输出流. `TBLPROPERTIES` 从句指定了一个 [KafkaProducer](#) 的配置, 这个从句是 Kafka sink 表必须的.

第三个语句是一个定义拓扑的 `SELECT` 语句: 它指示 StormSQL 过滤 `ORDERS` 表中的所有订单, 计算各订单总价并将匹配的记录插入 `LARGE_ORDER` 指定的 Kafka 流中.

要想运行这个例子, 用户需要在 `classpath` 中包含数据源 (本例中 `storm-sql-kafka`)和它的所有依赖. 当运行 `storm sql` 的时候 Storm SQL 的依赖会自动处理. 用户可以在提交的步骤中包含数据源依赖, 如下所示:

```
$ bin/storm sql order_filtering.sql order_filtering --artifacts "org.apache.storm:storm-sql-kafka:2.0.0"
```

上面的命令提交 SQL 语句到 StormSQL. 如果用户使用了不同版本的 Storm 或者 Kafka, 需要替换每个 `artifacts` 的版本.

现在, 应该能在 Storm UI 中看到 `order_filtering` 拓扑.

## 显示查询计划(explain mode)

就像 SQL 语句上的 `explain`, StormSQL 在运行 Storm SQL 执行器时提供 `explain mode`. 在分析模式下, StormSQL 分析每一个查询语句(仅DML)并显示执行计划而不是提交拓扑.

为了运行 `explain mode`, 需要设置拓扑名称为 `--explain` 并像用和提交相同的方式执行 `storm sql` 命令.

例如, 当以分析模式运行上面的例子的时:

```
$ bin/storm sql order_filtering.sql --explain --artifacts "org.apache.storm:storm-sql-kafka:2.0.0-SNAPSHOT"
```

StormSQL 输出打印如下:

```
=====
query>
CREATE EXTERNAL TABLE ORDERS (ID INT PRIMARY KEY, UNIT_PRICE INT, QUANTITY INT) LOCATION 'kafka://127.0.0.1:9092'
-----
16:53:43.951 [main] INFO o.a.s.s.r.DataSourcesRegistry - Registering scheme kafka with org.apache.storm
No plan presented on DDL
=====
```

```
=====
query>
CREATE EXTERNAL TABLE LARGE_ORDERS (ID INT PRIMARY KEY, TOTAL INT) LOCATION 'kafka://localhost:2181'
-----
No plan presented on DDL
=====
```

```
=====
query>
INSERT INTO LARGE_ORDERS SELECT ID, UNIT_PRICE * QUANTITY AS TOTAL FROM ORDERS WHERE UNIT_PRICE * QUANTITY > 50
-----
plan>
LogicalTableModify(table=[[LARGE_ORDERS]], operation=[INSERT], updateColumnList=[[TOTAL]], flattened=[true])
  LogicalProject(ID=[\$0], TOTAL=[*(\$1, \$2)]), id = 7
    LogicalFilter(condition=[>(*(\$1, \$2), 50)]), id = 6
      EnumerableTableScan(table=[[ORDERS]]), id = 5
=====
```

## 局限

- Windowing 尚未实现.
- 不支持聚合和连接 (待到流SQL 成熟)

# Storm SQL 示例

本文通过处理 Apache 日志的例子来展示如何使用 Storm SQL. 本文使用 "how-to" 风格书写, 因此可以根据步骤, 一步一步的学习如何使用 Storm SQL.

## 准备

本文假定 Apache Zookeeper、Apache Storm、Apache Kafka 都本地安装并且正确配置运行. 方便起见, 本文假设 Apache Kafka 0.10.0 是通过 brew 安装.

我们将使用下列工具为输入数据源生成 JSON 数据. 因为他们都是 Python 工程, 本页假设已经安装了 Python 2.7、`pip`、`virtualenv`. 如果你使用 Python 3, 需要在生成数据的时候修改一些与 Python 3 不兼容的代码.

- <https://github.com/kiritbasu/Fake-Apache-Log-Generator>
- <https://github.com/rory/apache-log-parser>

## 创建 topic

在本页, 我们会使用3个 topic, `apache-logs`, `apache-errorlogs`, `apache-slowlogs`. 请根据你的环境来创建 topic.

对于使用 brew 安装的 Apache Kafka 0.10.0,

```
kafka-topics --create --topic apache-logs --zookeeper localhost:2181 --replication-factor 1 --parti
kafka-topics --create --topic apache-errorlogs --zookeeper localhost:2181 --replication-factor 1 --
kafka-topics --create --topic apache-slowlogs --zookeeper localhost:2181 --replication-factor 1 --p
```

## 灌入数据

让我们提供数据给输入 topic. 在本页中我们将生成假的 Apache 日志, 转换为 JSON 格式, 并把 JSON 灌入 Kafka topic.

开始创建你的工作目录, 用于克隆项目并且设置 `virtualenv`.

在你的工作目录, 命令 `virtualenv env` 把 `env` 目录设置为 `virtualenv` 目录, 然后激活虚拟环境.

```
$ virtualenv env
$ source env/bin/activate
```

完成例子以后, 可以随时 `deactivate`, 退出 python 虚拟环境.

# 安装和修改 Fake-Apache-Log-Generator

Fake-Apache-Log-Generator 对包不可见, 我们还需要修改一下脚本.

```
$ git clone https://github.com/kiritbasu/Fake-Apache-Log-Generator.git  
$ cd Fake-Apache-Log-Generator
```

打开 apache-fake-log-gen.py, 将 while (flag): 语句替换成下面的语句:

```
elapsed_us = random.randint(1 * 1000,1000 * 1000) # 1 ms to 1 sec  
seconds=random.randint(30,300)  
increment = datetime.timedelta(seconds=seconds)  
otime += increment  
  
ip = faker.ipv4()  
dt = otime.strftime('%d/%b/%Y:%H:%M:%S')  
tz = datetime.datetime.now(pytz.timezone('US/Pacific')).strftime('%z')  
vrb = numpy.random.choice(verb,p=[0.6,0.1,0.1,0.2])  
  
uri = random.choice(resources)  
if uri.find("apps")>0:  
    uri += `random.randint(1000,10000)`  
  
resp = numpy.random.choice(response,p=[0.9,0.04,0.02,0.04])  
byt = int(random.gauss(5000,50))  
referer = faker.uri()  
useragent = numpy.random.choice(ualist,p=[0.5,0.3,0.1,0.05,0.05] )()  
f.write('%s -- [%s %s] %s "%s %s HTTP/1.0" %s %s "%s" "%s"\n' % (ip,dt,tz,elapsed_us,vrb,uri,log_lines - 1))  
flag = False if log_lines == 0 else True
```

要确保 elapsed\_us 包含在假日志中.

为了方便, 你可以跳过克隆项目这一步, 直接从这里下载修改过的文件: [apache-fake-log-gen.py \(gist\)](#)

## 安装 apache-log-parser 并编写转换脚本

apache-log-parser 模块可以通过 pip 命令安装.

```
$ pip install apache-log-parser
```

因为 apache-log-parser 是一个 python 库, 为了转换日志我们需要写编写一个小脚本. 我们创建一个文件 parse-fake-log-gen-to-json-with-incrementing-id.py 包含以下内容:

```
import sys  
import apache_log_parser
```

```
import json

auto_incr_id = 1
parser_format = '%a - - %t %D "%r" %s %b "%{Referer}i" "%{User-Agent}i"'
line_parser = apache_log_parser.make_parser(parser_format)
while True:
    # we'll use pipe
    line = sys.stdin.readline()
    if not line:
        break
    parsed_dict = line_parser(line)
    parsed_dict['id'] = auto_incr_id
    auto_incr_id += 1

    # works only python 2, but I don't care cause it's just a test module :)
    parsed_dict = {k.upper(): v for k, v in parsed_dict.iteritems() if not k.endswith('datetimeobj')}
    print json.dumps(parsed_dict)
```

## 将转换后的 **JSON Apache Log** 灌入 **Kafka**

好了! 我们已经准备好将数据写入 **Kafka topic**. 下面使用 `kafka-console-producer` 来灌入 **JSON**.

```
$ python apache-fake-log-gen.py -n 0 | python parse-fake-log-gen-to-json-with-incrementing-id.py |
```

打开另一个终端执行下面的命令, 确认数据已经进入 **topic**.

```
$ kafka-console-consumer --zookeeper localhost:2181 --topic apache-logs
```

如果看到如下的 **json**, 就说明搞定了:

```
{"TIME_US": "757467", "REQUEST_FIRST_LINE": "GET /wp-content HTTP/1.0", "REQUEST_METHOD": "GET", "R
```

## 例子: 过滤错误日志

在这个例子中, 我们将从所有的日志中过滤出 **error** 日志并且存储到另一个 **topic** 中. 将会用到 `project` 和 `filter` 特性.

脚本文件的内容如下:

```
CREATE EXTERNAL TABLE APACHE_LOGS (ID INT PRIMARY KEY, REMOTE_IP VARCHAR, REQUEST_URL VARCHAR, REQU
CREATE EXTERNAL TABLE APACHE_ERROR_LOGS (ID INT PRIMARY KEY, REMOTE_IP VARCHAR, REQUEST_URL VARCHAR
INSERT INTO APACHE_ERROR_LOGS SELECT ID, REMOTE_IP, REQUEST_URL, REQUEST_METHOD, CAST STATUS AS INT
```

把文件保存为 `apache_log_error_filtering.sql`.

让我们过一遍这个脚本.

第一个语句定义了一个表 `APACHE_LOGS` 代表输入流. `LOCATION` 从句指定了 `ZkHost` (`localhost:2181`), `brokers` 路径 (`/brokers`) 和 `topic` (`apache-logs`). 注意 `Kafka` 数据源必须定义一个主键. 这就是为什么我们为 `JSON` 数据设置了一个整数 `id`.

同样, 第二个语句指定了表 `APACHE_ERROR_LOGS` 代表输出流. `TBLPROPERTIES` 从句指定了 `KafkaProducer` 的配置, 从句对于 `Kafka sink` 表是必须的.

最后的语句定义了一个 `topology`. `Storm SQL` 只会在 `DML` 语句上定义和运行 `topology`. `DDL` 语句定义输入数据源、输出数据源、以及可以被 `DML` 语句引用的用户定义函数(`user defined function`).

我们先看 `where` 语句. 由于我们想过滤 `error` 日志, 我们使用状态码除以 `100`, 验证得到的商是否等于或者大于 `4`.(简单的说就是 `status_code >= 400`) 由于 `JSON` 中的状态码是字符串格式(因此在 `APACHE_LOGS` 表中是 `VARCHAR` 格式). 我们在应用除法之前, 使用 `CAST(status AS INT)` 先把状态码转换为整数. 现在我们只有 `error` 日志了.

我们转换一些列以和输出流想匹配. 在这个语句中, 我们使用 `CAST(status AS INT)` 转化为整数类型, 然后使用 `1000` 除 `TIME_US` 将毫秒转换成秒.

最后, `insert` 语句将过滤和转换后的行(`tuples`)存入输出流.

要运行这个例子, 用户需要包含数据源(本例中是 `storm-sql-kafka`) 和的所有依赖到 `class path` 中. 当用户运行 `storm sql` 命令的时候 `Storm SQL` 的依赖会被自动处理. 用户可以在提交阶段包含数据源依赖, 如下:

```
$ $STORM_DIR/bin/storm sql apache_log_error_filtering.sql apache_log_error_filtering --artifacts "o
```

上面的命令提交 `SQL` 语句到 `StormSQL`. `storm sql` 命令的选项是

`storm sql [script file] [topology name]`. 如果用户使用了不同版本的 `Storm` 或者 `Kafka`, 需要修改每个 `artifacts` 的版本号与之对应.

如果你的语句通过了验证阶段, 会在 `Storm UI` 页面上显示 `topology`.

你可以在控制台上看到下面的输出:

```
$ kafka-console-consumer --zookeeper localhost:2181 --topic apache-error-logs
```

输出类似下面的内容:

```
{"ID":854643,"REMOTE_IP":"4.227.214.159","REQUEST_URL":"/wp-content","REQUEST_METHOD":"GET","STATUS":  
{"ID":854693,"REMOTE_IP":"223.50.249.7","REQUEST_URL":"/apps/cart.jsp?appID=5578","REQUEST_METHOD":  
...}
```

你可以运行 **Storm SQL runner**, 将 **topology** 名称替换为 `--explain` 来查看逻辑执行计划.

```
$ $STORM_DIR/bin/storm sql apache_log_error_filtering.sql --explain --artifacts "org.apache.storm:s
```

输入类似下面的内容:

```
LogicalTableModify(table=[[APACHE_ERROR_LOGS]], operation=[INSERT], updateColumnList=[[ ]], flattened=  
LogicalProject(ID=[ $0 ], REMOTE_IP=[ $1 ], REQUEST_URL=[ $2 ], REQUEST_METHOD=[ $3 ], STATUS=[CAST($4):I  
LogicalFilter(condition=[>=(/($4):INTEGER NOT NULL, 100), 4]), id = 6  
EnumerableTableScan(table=[[APACHE_LOGS]]), id = 5
```

如果 **Storm SQL** 应用了查询优化, 你可能看到的输出会和上面的不一样.

我们正在执行第一个 **Storm SQL topology!** 如果你看到了足够多的输出和日志, 请杀掉 **topology**.

为了简洁, 我们不再解释我们已经看到的东西.

### 例子: 过滤访问慢的日志

在这个例子中我们将过滤访问慢的日志, 把他们存储到另一个 **topic**. 用到的特性有 `project`、`filter`、`User Defined Function (UDF)`. 这个例子与上一个例子 `filtering error logs` 非常相似, 我们主要看如何定义 `User Defined Function (UDF)`.

脚本文件的内容如下:

```
CREATE EXTERNAL TABLE APACHE_LOGS (ID INT PRIMARY KEY, REMOTE_IP VARCHAR, REQUEST_URL VARCHAR, REQU  
CREATE EXTERNAL TABLE APACHE_SLOW_LOGS (ID INT PRIMARY KEY, REMOTE_IP VARCHAR, REQUEST_URL VARCHAR,  
CREATE FUNCTION GET_TIME AS 'org.apache.storm.sql.runtime.functions.scalar.datetime.GetTime2'  
INSERT INTO APACHE_SLOW_LOGS SELECT ID, REMOTE_IP, REQUEST_URL, REQUEST_METHOD, CAST(STATUS AS INT)
```

内容保存为文件 `apache_log_slow_filtering.sql`.

由于前两个语句和上一个例子相似, 我们直接跳过.

第三个语句定义了一个 `User defined function`. 我们使用

org.apache.storm.sql.runtime.functions.scalar.datetime.GetTime2 定义了一个 GET\_TIME.

GetTime2 函数的实现如下：

```
package org.apache.storm.sql.runtime.functions.scalar.datetime;

import org.joda.time.format.DateTimeFormat;
import org.joda.time.format.DateTimeFormatter;

public class GetTime2 {
    public static Long evaluate(String dateString, String dateFormat) {
        try {
            DateTimeFormatter df = DateTimeFormat.forPattern(dateFormat).withZoneUTC();
            return df.parseDateTime(dateString).getMillis();
        } catch (Exception ex) {
            throw new RuntimeException(ex);
        }
    }
}
```

由于这个类定义了静态方法 evaluate 可以用于 UDF. SQL 的参数和返回的类型取决于 Storm SQL 依赖哪种风格.

注意, 这个类应该放在 classpath 路径下, 因此为了定义 UDF, 你需要创建一个包含 UDF 类的 jar 文件, 并在执行 storm sql 命令的时候使用 -- jar 选项.

最后一个语句和过滤错误日志的例子相似. 唯一新鲜的东西就是我们调用了

GET\_TIME(TIME\_RECEIVED\_UTC\_ISOFORMAT, 'yyyy-MM-dd''T''HH:mm:ssZZ') 将字符串格式的时间转换为 unix timestamp (BIGINT).

执行:

```
$ $STORM_DIR/bin/storm sql apache_log_slow_filtering.sql apache_log_slow_filtering --artifacts "org
```

可以在控制台看到下面的输出:

```
$ kafka-console-consumer --zookeeper localhost:2181 --topic apache-slow-logs
```

输出类似下面的内容:

```
{"ID":890502,"REMOTE_IP":"136.156.159.160","REQUEST_URL":"/list","REQUEST_METHOD": "GET", "USER_AGENT": "Mozilla/5.0 (Windows NT 5.01) AppleWebKit/5311 (KHTML, like Gecko) Chrome/13.0.860.0 Safari/5311","TIME_RECEIVED_UTC_ISOFORMAT":"2021-06-
```

05T03:44:59+00:00", "TIME\_RECEIVED\_TIMESTAMP": 1622864699000, "TIME\_ELAPSED\_  
{"ID": 890542, "REMOTE\_IP": "105.146.3.190", "REQUEST\_URL": "/search/tag/list", "REQUEST\_METHOD": "GET", "HTTP\_USER\_AGENT": "Mozilla/5.0 (X11; Linux i686) AppleWebKit/5332 (KHTML, like Gecko) Chrome/13.0.891.0  
Safari/5332", "TIME\_RECEIVED\_UTC\_ISOFORMAT": "2021-06-05T05:54:27+00:00", "TIME\_RECEIVED\_TIMESTAMP": 1622872467000, "TIME\_ELAPSED\_  
...  
...

好了！假设我们有通过远程 IP 查询 geo 信息的 UDF，我们可以通过 geo 位置做过滤，或者将 geo 位置添加到转换结果中。

## Summary

我们通读了几个 Storm SQL 的简单的用例来学习 Storm SQL 的特性. 如果还没有看过 [Storm SQL integration](#) 和 [Storm SQL language](#), 你需要阅读这些章节来查看所有支持的特性.

注意, Storm SQL 运行在小批量和非强类型的 Trident 库之上. Sink 实际并不检查类型. (你可能注意到一些输出字段的类型与输出表模式的定义不同).

当 Storm SQL 的后端 API 修改为核 (tuple by tuple, low-level, high-level) 时, Storm SQL 的行为会相应改变.

# Storm SQL 语言参考

Storm SQL 使用 Apache Calcite 来转换和评估 SQL 语句. Storm SQL 还采用了来自 Calcite 的 Rex 编译器, 因此 Storm SQL 将处理由 Calcite 的默认 SQL 解析器识别的 SQL 方言。

本文基于 Calcite 官网的 SQL 参考手册, 移除了部分 Storm SQL 不支持的内容, 添加了一些 Storm SQL 支持的内容.

请先阅读 [Storm SQL integration](#) 页面, 了解 Storm SQL 支持哪些特性.

## 语法

Calcite 提供丰富的 SQL 语法. 但是 Storm SQL 并不是一个数据库系统, 它用于处理流式数据, 因此只支持语法的子集. Storm SQL 不会重定义 SQL 语法, 只优化 Calcite 提供的转换器, 因此 SQL 语句仍然基于 Calcite 的 SQL 语法进行转换.

SQL 语法表现为类BNF 的形式.

```
statement:
  setStatement
  resetStatement
  explain
  describe
  insert
  update
  merge
  delete
  query

setStatement:
  [ ALTER ( SYSTEM | SESSION ) ] SET identifier '=' expression

resetStatement:
  [ ALTER ( SYSTEM | SESSION ) ] RESET identifier
  | [ ALTER ( SYSTEM | SESSION ) ] RESET ALL

explain:
  EXPLAIN PLAN
  [ WITH TYPE | WITH IMPLEMENTATION | WITHOUT IMPLEMENTATION ]
  [ EXCLUDING ATTRIBUTES | INCLUDING [ ALL ] ATTRIBUTES ]
  FOR ( query | insert | update | merge | delete )

describe:
  DESCRIBE DATABASE databaseName
  | DESCRIBE CATALOG [ databaseName . ] catalogName
  | DESCRIBE SCHEMA [ [ databaseName . ] catalogName ] . schemaName
  | DESCRIBE [ TABLE ] [ [ [ databaseName . ] catalogName . ] schemaName . ] tableName [ columnNa
  | DESCRIBE [ STATEMENT ] ( query | insert | update | merge | delete )

insert:
```

```
( INSERT | UPSERT ) INTO tablePrimary
[ '(' column [, column ]* ')' ]
query

update:
  UPDATE tablePrimary
  SET assign [, assign ]*
  [ WHERE booleanExpression ]

assign:
  identifier '=' expression

merge:
  MERGE INTO tablePrimary [ [ AS ] alias ]
  USING tablePrimary
  ON booleanExpression
  [ WHEN MATCHED THEN UPDATE SET assign [, assign ]* ]
  [ WHEN NOT MATCHED THEN INSERT VALUES '(' value [, value ]* ')' ]

delete:
  DELETE FROM tablePrimary [ [ AS ] alias ]
  [ WHERE booleanExpression ]

query:
  values
  | WITH withItem [ , withItem ]* query
  | {
  |   select
  |   selectWithoutFrom
  |   query UNION [ ALL ] query
  |   query EXCEPT query
  |   query INTERSECT query
  | }
  | [ ORDER BY orderItem [, orderItem ]* ]
  | [ LIMIT { count | ALL } ]
  | [ OFFSET start { ROW | ROWS } ]
  | [ FETCH { FIRST | NEXT } { count } { ROW | ROWS } ]

withItem:
  name
  [ '(' column [, column ]* ')' ]
  AS '(' query ')'

orderItem:
  expression [ ASC | DESC ] [ NULLS FIRST | NULLS LAST ]

select:
  SELECT [ STREAM ] [ ALL | DISTINCT ]
    { * | projectItem [, projectItem ]* }
  FROM tableExpression
  [ WHERE booleanExpression ]
  [ GROUP BY { groupItem [, groupItem ]* } ]
  [ HAVING booleanExpression ]
  [ WINDOW windowName AS windowSpec [, windowName AS windowSpec ]* ]

selectWithoutFrom:
  SELECT [ ALL | DISTINCT ]
    { * | projectItem [, projectItem ]* }
```

```

projectItem:
    expression [ [ AS ] columnAlias ]
  | tableAlias . *

tableExpression:
    tableReference [, tableReference ]*
  | tableExpression [ NATURAL ] [ LEFT | RIGHT | FULL ] JOIN tableExpression [ joinCondition ]

joinCondition:
    ON booleanExpression
  | USING '(' column [ , column ]* ')'

tableReference:
    tablePrimary
    [ [ AS ] alias [ '(' columnAlias [ , columnAlias ]* ')' ] ]

tablePrimary:
    [ [ catalogName . ] schemaName . ] tableName
    '(' TABLE [ [ catalogName . ] schemaName . ] tableName ')'
  | [ LATERAL ] '(' query ')'
  | UNNEST '(' expression ')' [ WITH ORDINALITY ]
  | [ LATERAL ] TABLE '(' [ SPECIFIC ] functionName '(' expression [ , expression ]* ')' ')'

values:
    VALUES expression [ , expression ]*

groupItem:
    expression
    '(' ')'
    '(' expression [ , expression ]* ')'
    CUBE '(' expression [ , expression ]* ')'
    ROLLUP '(' expression [ , expression ]* ')'
    GROUPING SETS '(' groupItem [ , groupItem ]* ')'

windowRef:
    windowName
  | windowSpec

windowSpec:
    [ windowName ]
    '('
    [ ORDER BY orderItem [ , orderItem ]* ]
    [ PARTITION BY expression [ , expression ]* ]
    [
        RANGE numericOrIntervalExpression { PRECEDING | FOLLOWING }
      | ROWS numericExpression { PRECEDING | FOLLOWING }
    ]
    ')'

```

在 `merge` 中, 必须有 `WHEN MATCH` 或者 `WHEN NOT MATCH` 从句的其中之一.

在 `orderItem` 中, 如果 `expression` 是一个正整数 `n`, 他表示 `SELECT` 从句中的第 `n` 个项目.

聚合查询是包含 GROUP BY 或 HAVING 的查询, 子句或 SELECT 子句中的聚合函数。在 SELECT 中, 汇总查询的 HAVING 和 ORDER BY 子句, 所有表达式必须在当前组内是恒定的(即分组常量由 GROUP BY 子句定义, 或常量)或聚合函数或常量和聚合的组合功能。聚合和分组功能只能出现在聚合查询, 并且仅在 SELECT, HAVING 或 ORDER BY 子句中。

一个标量子查询是像表达式一样使用的子查询. 如果子查询没有返回行, 值为 NULL; 如果返回多行, 就是错误的.

IN, EXISTS 和 标量子查询可以在任何使用 expression 的地方使用(比如 SELECT 从句, WHERE 从句, JOIN 后面的 ON 从句, 或者作为一个聚合函数的参数).

一个 IN, EXISTS 或者标量子查询可能是相关的; 意思是, 可以引用封闭查询的 FROM 从句中的表。

*selectWithoutFrom* 等价于 VALUES, 但是并非标准 SQL, 只在允许在特定的 [conformance levels](#) 上.

## 关键词

下列是一个 SQL 的关键词列表. 这个列表页来自于 Calcite SQL 的参考手册.

A, **ABS**, ABSOLUTE, ACTION, ADA, ADD, ADMIN, AFTER, **ALL**, **ALLOCATE**, **ALLOW**,  
**ALTER**, ALWAYS, **AND**, **ANY**, **ARE**, **ARRAY**, **AS**, ASC, **ASENSITIVE**, ASSERTION,  
ASSIGNMENT, **ASYMMETRIC**, AT, **ATOMIC**, ATTRIBUTE, ATTRIBUTES,  
**AUTHORIZATION**, **AVG**, BEFORE, **BEGIN**, BERNOULLI, **BETWEEN**, **BIGINT**, **BINARY**,  
**BIT**, **BLOB**, **BOOLEAN**, **BOTH**, BREADTH, **BY**, C, **CALL**, **CALLED**, **CARDINALITY**,  
CASCADE, **CASCADED**, **CASE**, **CAST**, CATALOG, CATALOG\_NAME, **CEIL**, **CEILING**,  
CENTURY, CHAIN, **CHAR**, **CHARACTER**, CHARACTERISTICS, CHARACTERS,  
**CHARACTER\_LENGTH**, CHARACTER\_SET\_CATALOG, CHARACTER\_SET\_NAME,  
CHARACTER\_SET\_SCHEMA, **CHAR\_LENGTH**, **CHECK**, CLASS\_ORIGIN, **CLOB**,  
**CLOSE**, **COALESCE**, COBOL, **COLLATE**, COLLATION, COLLATION\_CATALOG,  
COLLATION\_NAME, COLLATION\_SCHEMA, **COLLECT**, **COLUMN**, COLUMN\_NAME,  
COMMAND\_FUNCTION, COMMAND\_FUNCTION\_CODE, **COMMIT**, COMMITTED,  
**CONDITION**, CONDITION\_NUMBER, **CONNECT**, CONNECTION, CONNECTION\_NAME,  
**CONSTRAINT**, CONSTRAINTS, CONSTRAINT\_CATALOG, CONSTRAINT\_NAME,

CONSTRAINT\_SCHEMA, CONSTRUCTOR, CONTAINS, CONTINUE, CONVERT, CORR,  
CORRESPONDING, COUNT, COVAR\_POP, COVAR\_SAMP, CREATE, CROSS, CUBE,  
CUME\_DIST, CURRENT, CURRENT\_CATALOG, CURRENT\_DATE,  
CURRENT\_DEFAULT\_TRANSFORM\_GROUP, CURRENT\_PATH, CURRENT\_ROLE,  
CURRENT\_SCHEMA, CURRENT\_TIME, CURRENT\_TIMESTAMP,  
CURRENT\_TRANSFORM\_GROUP\_FOR\_TYPE, CURRENT\_USER, CURSOR,  
CURSOR\_NAME, CYCLE, DATA, DATABASE, DATE, DATETIME\_INTERVAL\_CODE,  
DATETIME\_INTERVAL\_PRECISION, DAY, DEALLOCATE, DEC, DECADE, DECIMAL,  
DECLARE, DEFAULT, DEFAULTS, DEFERRABLE, DEFERRED, DEFINED, DEFINER,  
DEGREE, DELETE, DENSE\_RANK, DEPTH, DEREF, DERIVED, DESC, DESCRIBE,  
DESCRIPTION, DESCRIPTOR, DETERMINISTIC, DIAGNOSTICS, DISALLOW,  
DISCONNECT, DISPATCH, DISTINCT, DOMAIN, DOUBLE, DOW, DOY, DROP,  
DYNAMIC, DYNAMIC\_FUNCTION, DYNAMIC\_FUNCTION\_CODE, EACH, ELEMENT,  
ELSE, END, END-EXEC, EPOCH, EQUALS, ESCAPE, EVERY, EXCEPT, EXCEPTION,  
EXCLUDE, EXCLUDING, EXEC, EXECUTE, EXISTS, EXP, EXPLAIN, EXTEND,  
EXTERNAL, EXTRACT, FALSE, FETCH, FILTER, FINAL, FIRST, FIRST\_VALUE, FLOAT,  
FLOOR, FOLLOWING, FOR, FOREIGN, FORTRAN, FOUND, FRAC\_SECOND, FREE,  
FROM, FULL, FUNCTION, FUSION, G, GENERAL, GENERATED, GET, GLOBAL, GO,  
GOTO, GRANT, GRANTED, GROUP, GROUPING, HAVING, HIERARCHY, HOLD,  
HOUR, IDENTITY, IMMEDIATE, IMPLEMENTATION, IMPORT, IN, INCLUDING,  
INCREMENT, INDICATOR, INITIALLY, INNER, INOUT, INPUT, INSENSITIVE, INSERT,  
INSTANCE, INSTANTIABLE, INT, INTEGER, INTERSECT, INTERSECTION, INTERVAL,  
INTO, INVOKER, IS, ISOLATION, JAVA, JOIN, K, KEY, KEY\_MEMBER, KEY\_TYPE,  
LABEL, LANGUAGE, LARGE, LAST, LAST\_VALUE, LATERAL, LEADING, LEFT,  
LENGTH, LEVEL, LIBRARY, LIKE, LIMIT, LN, LOCAL, LOCALTIME,  
LOCALTIMESTAMP, LOCATOR, LOWER, M, MAP, MATCH, MATCHED, MAX,  
MAXVALUE, MEMBER, MERGE, MESSAGE\_LENGTH, MESSAGE\_OCTET\_LENGTH,  
MESSAGE\_TEXT, METHOD, MICROSECOND, MILLENIUM, MIN, MINUTE, MINVALUE,  
MOD, MODIFIES, MODULE, MONTH, MORE, MULTISET, MUMPS, NAME, NAMES,  
NATIONAL, NATURAL, NCHAR, NCLOB, NESTING, NEW, NEXT, NO, NONE,  
NORMALIZE, NORMALIZED, NOT, NULL, NULLABLE, NULLIF, NULLS, NUMBER,  
NUMERIC, OBJECT, OCTETS, OCTET\_LENGTH, OF, OFFSET, OLD, ON, ONLY,

OPEN, OPTION, OPTIONS, OR, ORDER, ORDERING, ORDINALITY, OTHERS, OUT, OUTER, OUTPUT, OVER, OVERLAPS, OVERLAY, OVERRIDING, PAD, PARAMETER, PARAMETER\_MODE, PARAMETER\_NAME, PARAMETER\_ORDINAL\_POSITION, PARAMETER\_SPECIFIC\_CATALOG, PARAMETER\_SPECIFIC\_NAME, PARAMETER\_SPECIFIC\_SCHEMA, PARTIAL, PARTITION, PASCAL, PASSTHROUGH, PATH, PERCENTILE\_CONT, PERCENTILE\_DISC, PERCENT\_RANK, PLACING, PLAN, PLI, POSITION, POWER, PRECEDING, PRECISION, PREPARE, PRESERVE, PRIMARY, PRIOR, PRIVILEGES, PROCEDURE, PUBLIC, QUARTER, RANGE, RANK, READ, READS, REAL, RECURSIVE, REF, REFERENCES, REFERENCING, REGR\_AVGX, REGR\_AVGY, REGR\_COUNT, REGR\_INTERCEPT, REGR\_R2, REGR\_SLOPE, REGR\_SXX, REGR\_SXY, REGR\_SYY, RELATIVE, RELEASE, REPEATABLE, RESET, RESTART, RESTRICT, RESULT, RETURN, RETURNED\_CARDINALITY, RETURNED\_LENGTH, RETURNED\_OCTET\_LENGTH, RETURNED\_SQLSTATE, RETURNS, REVOKE, RIGHT, ROLE, ROLLBACK, ROLLUP, ROUTINE, ROUTINE\_CATALOG, ROUTINE\_NAME, ROUTINE\_SCHEMA, ROW, ROWS, ROW\_COUNT, ROW\_NUMBER, SAVEPOINT, SCALE, SCHEMA, SCHEMA\_NAME, SCOPE, SCOPE\_CATALOGS, SCOPE\_NAME, SCOPE\_SCHEMA, SCROLL, SEARCH, SECOND, SECTION, SECURITY, SELECT, SELF, SENSITIVE, SEQUENCE, SERIALIZABLE, SERVER, SERVER\_NAME, SESSION, SESSION\_USER, SET, SETS, SIMILAR, SIMPLE, SIZE, SMALLINT, SOME, SOURCE, SPACE, SPECIFIC, SPECIFICTYPE, SPECIFIC\_NAME, SQL, SQLEXCEPTION, SQLSTATE, SQLWARNING, SQL\_TSI\_DAY, SQL\_TSI\_FRAC\_SECOND, SQL\_TSI\_HOUR, SQL\_TSI\_MICROSECOND, SQL\_TSI\_MINUTE, SQL\_TSI\_MONTH, SQL\_TSI\_QUARTER, SQL\_TSI\_SECOND, SQL\_TSI\_WEEK, SQL\_TSI\_YEAR, SQRT, START, STATE, STATEMENT, STATIC, STDDEV\_POP, STDDEV\_SAMP, STREAM, STRUCTURE, STYLE, SUBCLASS\_ORIGIN, SUBMULTISET, SUBSTITUTE, SUBSTRING, SUM, SYMMETRIC, SYSTEM, SYSTEM\_USER, TABLE, TABLESAMPLE, TABLE\_NAME, TEMPORARY, THEN, TIES, TIME, TIMESTAMP, TIMESTAMPADD, TIMESTAMPDIFF, TIMEZONE\_HOUR, TIMEZONE\_MINUTE, TINYINT, TO, TOP\_LEVEL\_COUNT, TRAILING, TRANSACTION, TRANSACTIONS\_ACTIVE, TRANSACTIONS\_COMMITED, TRANSACTIONS\_ROLLED\_BACK, TRANSFORM, TRANSFORMS, TRANSLATE, TRANSLATION, TREAT, TRIGGER, TRIGGER\_CATALOG, TRIGGER\_NAME,

**TRIGGER\_SCHEMA, TRIM, TRUE, TYPE, UESCAPE, UNBOUNDED, UNCOMMITTED,**  
**UNDER, UNION, UNIQUE, UNKNOWN, UNNAMED, UNNEST, UPDATE, UPPER,**  
**UPSERT, USAGE, USER, USER\_DEFINED\_TYPE\_CATALOG,**  
**USER\_DEFINED\_TYPE\_CODE, USER\_DEFINED\_TYPE\_NAME,**  
**USER\_DEFINED\_TYPE\_SCHEMA, USING, VALUE, VALUES, VARBINARY, VARCHAR,**  
**VARYING, VAR\_POP, VAR\_SAMP, VERSION, VIEW, WEEK, WHEN, WHENEVER,**  
**WHERE, WIDTH\_BUCKET, WINDOW, WITH, WITHIN, WITHOUT, WORK, WRAPPER,**  
**WRITE, XML, YEAR, ZONE.**

## 标识符

标识符是在 SQL 查询中使用的表名, 列, 和其他元数据元素.

未被引号括起来的标识符, 比如 `emp`, 必须以字母打头且只能包含字母, 数字, 下划线. 他们会隐式的转换为大写.

被引号引起的标识符, 例如 `"Employee Name"`, 以双引号开始和结束. 他们可以包含几乎任何字符, 包括空白和其他标点. 如果你想在标识符中包含一个双引号, 使用双引号进行转义, 像这样: `"An employee called ""Fred""."`.

在 Calcite 中, 与引用的对象的名称匹配的标识符是大小写敏感的. 但是记住, 被引号括起来的标识符会在匹配前隐式转化为大写, 如果它引用的对象的名称是使用未被引号括起来的标识符创建的, 它的名称也会被转换成大写.

## 数据类型

### 标量类型

| Data type    | Description           | Range and examples                 |
|--------------|-----------------------|------------------------------------|
| BOOLEAN      | Logical values        | Values: TRUE, FALSE, UNKNOWN       |
| TINYINT      | 1 byte signed integer | Range is -255 to 256               |
| SMALLINT     | 2 byte signed integer | Range is -32768 to 32767           |
| INTEGER, INT | 4 byte signed         | Range is -2147483648 to 2147483647 |

|                                   |                                  |                                                                                                                         |
|-----------------------------------|----------------------------------|-------------------------------------------------------------------------------------------------------------------------|
|                                   | integer                          |                                                                                                                         |
| BIGINT                            | 8 byte signed integer            | Range is -9223372036854775808 to 9223372036854775807                                                                    |
| DECIMAL(p, s)                     | Fixed point                      | Example: 123.45 is a DECIMAL(5, 2) value.                                                                               |
| NUMERIC                           | Fixed point                      |                                                                                                                         |
| REAL, FLOAT                       | 4 byte floating point            | 6 decimal digits precision                                                                                              |
| DOUBLE                            | 8 byte floating point            | 15 decimal digits precision                                                                                             |
| CHAR(n), CHARACTER(n)             | Fixed-width character string     | 'Hello', " (empty string), _latin1'Hello', n'Hello', _UTF16'Hello', 'Hello' 'there' (literal split into multiple parts) |
| VARCHAR(n), CHARACTER VARYING(n)  | Variable-length character string | As CHAR(n)                                                                                                              |
| BINARY(n)                         | Fixed-width binary string        | x'45F0AB', x" (empty binary string), x'AB' 'CD' (multi-part binary string literal)                                      |
| VARBINARY(n), BINARY VARYING(n)   | Variable-length binary string    | As BINARY(n)                                                                                                            |
| DATE                              | Date                             | Example: DATE '1969-07-20'                                                                                              |
| TIME                              | Time of day                      | Example: TIME '20:17:40'                                                                                                |
| TIMESTAMP [ WITHOUT TIME ZONE ]   | Date and time                    | Example: TIMESTAMP '1969-07-20 20:17:40'                                                                                |
| TIMESTAMP WITH TIME ZONE          | Date and time with time zone     | Example: TIMESTAMP '1969-07-20 20:17:40 America/Los Angeles'                                                            |
| INTERVAL timeUnit [ TO timeUnit ] | Date time interval               | Examples: INTERVAL '1:5' YEAR TO MONTH, INTERVAL '45' DAY                                                               |

Anchored interval

Date time  
interval

Example: (DATE '1969-07-20', DATE '1972-08-29')

Where:

timeUnit:

MILLENNIUM | CENTURY | DECADE | YEAR | QUARTER | MONTH | WEEK | DOY | DOW | DAY | HOUR | MINUTE |

注意:

- DATE, TIME, TIMESTAMP 是不带时区的. 也没有比如UTC(像Java那样)或者本地时区作为默认时区. 它需要用户或者应用提供时区的处理.

非标量类型

| 类型       | 描述               |
|----------|------------------|
| ANY      | 一个类型未知的值         |
| ROW      | 一列或者多列组成的行       |
| MAP      | 键值映射的集合          |
| MULTISET | 可能包含重复内容的未排序的集合  |
| ARRAY    | 有序的, 可能包含重复的连续集合 |
| CURSOR   | 查询结果的游标          |

运算符和函数

运算符优先级

运算符优先级和结合性, 从高到低.

| 运算符                     | 优先级 |
|-------------------------|-----|
| .                       | 左   |
|                         | 左   |
| + - (unary plus, minus) | 右   |
| * /                     | 左   |

|                                     |   |
|-------------------------------------|---|
| + -                                 | 左 |
| BETWEEN, IN, LIKE, SIMILAR          | - |
| < > = <= >= <> !=                   | 左 |
| IS NULL, IS FALSE, IS NOT TRUE etc. | - |
| NOT                                 | 右 |
| AND                                 | 左 |
| OR                                  | 左 |

## 比较运算符

| 语法                                      | 描述                                                                                 |
|-----------------------------------------|------------------------------------------------------------------------------------|
| value1 = value2                         | 相等                                                                                 |
| value1 <> value2                        | 不等                                                                                 |
| value1 != value2                        | 不等 (only available at some conformance levels)                                     |
| value1 > value2                         | 大于                                                                                 |
| value1 >= value2                        | 大于等于                                                                               |
| value1 < value2                         | 小于                                                                                 |
| value1 <= value2                        | 小于等于                                                                               |
| value IS NULL                           | value 是否为 null                                                                     |
| value IS NOT NULL                       | value 是否不为 null                                                                    |
| value1 IS DISTINCT FROM<br>value2       | 两个值是否不等, null 值认为是相等                                                               |
| value1 IS NOT DISTINCT<br>FROM value2   | 两个值是否相等, null 值认为是相等                                                               |
| value1 BETWEEN value2 AND<br>value3     | Whether value1 is greater than or equal to value2 and less than or equal to value3 |
| value1 NOT BETWEEN value2<br>AND value3 | value1 是否小于 value2 或大于 value3                                                      |
| string1 LIKE string2 [ ESCAPE           | string1 与模式 string2 是否匹配                                                           |

string3 ]

string1 NOT LIKE string2 [  
ESCAPE string3 ]

*string1* 与模式 *string2* 是否不匹配

string1 SIMILAR TO string2 [  
ESCAPE string3 ]

*string1* 与正则 *string2* 是否匹配

string1 NOT SIMILAR TO  
string2 [ ESCAPE string3 ]

*string1* 与正则 *string2* 是否不匹配

value IN (value [, value]\* )

*value* 是否与列表中的每一个值都相等

value NOT IN (value [, value]\* )

*value* 是否与列表中的每一个值都不等

Not supported yet on Storm SQL: 目前 Storm SQL 中不支持的运算符:

| 语法                       | 描述                                        |
|--------------------------|-------------------------------------------|
| value IN (sub-query)     | 是否 <i>value</i> 等于 <i>sub-query</i> 返回的行  |
| value NOT IN (sub-query) | 是否 <i>value</i> 不等于 <i>sub-query</i> 返回的行 |
| EXISTS (sub-query)       | 是否 <i>sub-query</i> 返回至少一个行               |

Storm SQL 当前不支持子查询, 因此上面的操作不能正常工作. 这个问题会在不久的将来修复.

## 逻辑运算符

| 语法                    | 描述                                              |
|-----------------------|-------------------------------------------------|
| boolean1 OR boolean2  | 或                                               |
| boolean1 AND boolean2 | 且                                               |
| NOT boolean           | 是否为True ; <i>boolean</i> 为 UNKNOWN 则返回 UNKNOWN  |
| boolean IS FALSE      | 是否为False; <i>boolean</i> 为 UNKNOWN 则返回 UNKNOWN  |
| boolean IS NOT FALSE  | 是否不为False; <i>boolean</i> 为 UNKNOWN 则返回 UNKNOWN |
| boolean IS TRUE       | 是否为True; <i>boolean</i> 为 UNKNOWN 则返回 UNKNOWN   |
|                       | 是否不为True; <i>boolean</i> 为 UNKNOWN 则返回          |

|                        |                |
|------------------------|----------------|
| boolean IS NOT TRUE    | UNKNOWN        |
| boolean IS UNKNOWN     | 判断是否是 UNKNOWN  |
| boolean IS NOT UNKNOWN | 判断是否不是 UNKNOWN |

## 数学运算符和函数

| 语法                           | 描述                                                     |
|------------------------------|--------------------------------------------------------|
| + numeric                    | 返回 <i>numeric</i>                                      |
| :- numeric                   | 返回负 <i>numeric</i>                                     |
| numeric1 + numeric2          | 返回 <i>numeric1</i> 加 <i>numeric2</i>                   |
| numeric1 - numeric2          | 返回 <i>numeric1</i> 减 <i>numeric2</i>                   |
| numeric1 * numeric2          | 返回 <i>numeric1</i> 乘以 <i>numeric2</i>                  |
| numeric1 / numeric2          | 返回 <i>numeric1</i> 除以 <i>numeric2</i>                  |
| POWER(numeric1,<br>numeric2) | 返回 <i>numeric1</i> 的 <i>numeric2</i> 次方                |
| ABS(numeric)                 | 返回绝对值 <i>numeric</i>                                   |
| MOD(numeric, numeric)        | 取余 <i>numeric1</i> 除以 <i>numeric2</i> . 如果被除数为负, 则余数为负 |
| SQRT(numeric)                | 返回平方根 <i>numeric</i>                                   |
| LN(numeric)                  | 返回 <i>numeric</i> 的自然对数 (底数 e)                         |
| LOG10(numeric)               | 返回 <i>numeric</i> 的常用对数 (底数 10)                        |
| EXP(numeric)                 | 返回 e 的 <i>numeric</i> 次方                               |
| CEIL(numeric)                | 向上取整 <i>numeric</i> , 返回大于或者等于 <i>numeric</i> 的最小整数    |
| FLOOR(numeric)               | 向下取整 <i>numeric</i> , 返回小于或者等于 <i>numeric</i> 的最小整数    |

## 字符串运算符和函数

| 语法                                                             | 描述                                                    |
|----------------------------------------------------------------|-------------------------------------------------------|
| string    string                                               | 连接2个字符串                                               |
| CHAR_LENGTH(string)                                            | 返回字符串长度                                               |
| CHARACTER_LENGTH(string)                                       | 与 CHAR_LENGTH( <i>string</i> ) 等价                     |
| UPPER(string)                                                  | 转换为大写                                                 |
| LOWER(string)                                                  | 转换为小写                                                 |
| POSITION(string1 IN string2)                                   | 返回 <i>string1</i> 在 <i>string2</i> 中首次出现位置            |
| TRIM( { BOTH   LEADING   TRAILING } string1 FROM string2)      | 从 <i>string2</i> 的 头/尾/两头 移除 <i>string1</i> 的最长匹配     |
| OVERLAY(string1 PLACING string2 FROM integer [ FOR integer2 ]) | 用 <i>string2</i> 替换 <i>string1</i>                    |
| SUBSTRING(string FROM integer)                                 | 从给定的位置开始返回一个子串                                        |
| SUBSTRING(string FROM integer FOR integer)                     | 从给定位置返回一个指定长度的子串                                      |
| INITCAP(string)                                                | 将 <i>string</i> 中每个单词首字母大写其他字母小写.<br>单词是由空白字符分隔开的字符序列 |

未实现的:

- SUBSTRING(string FROM regexp FOR regexp)

## 二进制字符串操作符和函数

| 语法                                                             | 描述                                             |
|----------------------------------------------------------------|------------------------------------------------|
| binary    binary                                               | 连接2个二进制字符串.                                    |
| POSITION(binary1 IN binary2)                                   | 返回二进制串 <i>binary1</i> 在 <i>binary2</i> 中首次出现位置 |
| OVERLAY(binary1 PLACING binary2 FROM integer [ FOR integer2 ]) | 替换                                             |

已知的 bug:

## 语法

## 描述

SUBSTRING(binary FROM integer)

从指定的位置截取

SUBSTRING(binary FROM integer FOR integer)

从指定位置截取指定长度的串

Calcite 1.9.0 有bug, 当编译 SUBSTRING 函数的时候会抛出异常. 这个问题会在后续版本中修复.

## Date/time 函数

### 语法

### 描述

EXTRACT(timeUnit FROM datetime)

返回时间中的指定字段

FLOOR(datetime TO timeUnit)

根据 timeUnit 向下取整

CEIL(datetime TO timeUnit)

根据 timeUnit 向上取整

未实现的:

- EXTRACT(timeUnit FROM interval)
- CEIL(interval)
- FLOOR(interval)
- datetime - datetime timeUnit [ TO timeUnit ]
- interval OVERLAPS interval
- + interval
- - interval
- interval + interval
- interval - interval
- interval / interval
- datetime + interval
- datetime - interval

Storm SQL 特有的:

### 语法

### 描述

LOCALTIME

以类型 TIME 返回当前会话时区的当前时间

|                           |                                            |
|---------------------------|--------------------------------------------|
| LOCALTIME(precision)      | 以类型 TIME 返回当前会话时区的当前时间, 精度 precision       |
| LOCALTIMESTAMP            | 以类型 TIMESTAMP 返回当前会话时区的当前时间                |
| LOCALTIMESTAMP(precision) | 以类型 TIMESTAMP 返回当前会话时区的当前时间, 精度precision   |
| CURRENT_TIME              | 以类型 TIMESTAMP WITH TIME ZONE 返回当前会话时区的当前时间 |
| CURRENT_DATE              | 以类型 DATE 返回当前会话时区的当前时间                     |
| CURRENT_TIMESTAMP         | 以类型 TIMESTAMP WITH TIME ZONE 返回当前会话时区的当前时间 |

SQL标准规定, 上述运算符在评估查询时应返回相同的值。 Storm SQL将每个查询转换为 Trident拓扑并运行, 因此在评估SQL语句时, 技术上当前的日期/时间应该是固定的。 由于这个限制, 当创建Trident拓扑时, 当前日期/时间将被修复, 并且这些运算符应该在拓扑生命周期中返回相同的值。

## 系统函数

Storm SQL 当前不支持的函数:

| 语法           | 描述                             |
|--------------|--------------------------------|
| USER         | 等价于CURRENT_USER                |
| CURRENT_USER | 当前执行上下文的用户名                    |
| SESSION_USER | 会话的用户名                         |
| SYSTEM_USER  | 返回系统标识的当前数据存储的用户 user          |
| CURRENT_PATH | 返回表示当前查找范围的字符串, 用于引用用户定义的例程和类型 |
| CURRENT_ROLE | 返回当前活动 role                    |

这些操作符并不代表 Storm SQL 的运行时, 所以除非我们找到正确的语义, 否则这些操作可能永远不会被支持。

## 条件函数和操作符

## 语法

## 描述

CASE value  
WHEN value1 [, value11 ]\* THEN result1  
[ WHEN valueN [, valueN1 ]\* THEN  
resultN ]\*  
[ ELSE resultZ ]  
END

简单 case 语句

CASE  
WHEN condition1 THEN result1  
[ WHEN conditionN THEN resultN ]\*  
[ ELSE resultZ ]  
END

搜索 case 语句

NULLIF(value, value)

值相同返回.

例如, `NULLIF(5, 5)` 返回 `NULL`; `NULLIF(5, 0)` 返回 `5`.

COALESCE(value, value [, value ]\* )

前一个值为 `null` 则返回后一个值.

例如, `COALESCE(NULL, 5)` 返回 `5`.

## 类型转换

### 语法

### 描述

CAST(value AS type)

把值转换为给定的类型.

## 值构造器

### 语法

### 描述

ROW (value [, value]\* )

从值列表中创建一个行.

map '[' key ']

根据 `key` 返回 `value`.

array '[' index ']

返回某个位置的 `array` 的元素值.

ARRAY '[' value [, value ]\* ']

从值列表中创建一个 `array`.

MAP '[' key, value [, key, value ]\* ']

从 `key-value` 列表中创建一个 `map`.

## 集合函数

| 语法                          | 描述                                                          |
|-----------------------------|-------------------------------------------------------------|
| ELEMENT( <i>value</i> )     | 返回 array 或 multiset 的唯一元素; 如果集合为空, 则为null; 如果它有多个元素, 则抛出异常。 |
| CARDINALITY( <i>value</i> ) | 返回 array 或 multiset 的元素数.                                   |

另请参考: UNNEST关系运算符将集合转换为关系.

## JDBC 函数转义

### 数字

| 语法                                                 | 描述                                                                        |
|----------------------------------------------------|---------------------------------------------------------------------------|
| {fn ABS( <i>numeric</i> )}                         | 返回 <i>numeric</i> 的绝对值                                                    |
| {fn EXP( <i>numeric</i> )}                         | 返回 e 的 <i>numeric</i> 次方                                                  |
| {fn LOG( <i>numeric</i> )}                         | 返回 <i>numeric</i> 的自然对数 (底数为 e)                                           |
| {fn LOG10( <i>numeric</i> )}                       | 返回 <i>numeric</i> 的以10为底的对数                                               |
| {fn MOD( <i>numeric1</i> ,<br><i>numeric2</i> )}   | 返回 <i>numeric1</i> 被 <i>numeric2</i> 除的余数. 被除数 <i>numeric1</i> 为负数的时候余数为负 |
| {fn POWER( <i>numeric1</i> ,<br><i>numeric2</i> )} | 返回 <i>numeric1</i> 的 <i>numeric2</i> 次方                                   |

未实现的:

- {fn ACOS(*numeric*)} - Returns the arc cosine of *numeric*
- {fn ASIN(*numeric*)} - Returns the arc sine of *numeric*
- {fn ATAN(*numeric*)} - Returns the arc tangent of *numeric*
- {fn ATAN2(*numeric*, *numeric*)}
- {fn CEILING(*numeric*)} - Rounds *numeric* up, and returns the smallest number that is greater than or equal to *numeric*
- {fn COS(*numeric*)} - Returns the cosine of *numeric*
- {fn COT(*numeric*)}

- {fn DEGREES(numeric)} - Converts *numeric* from radians to degrees
- {fn FLOOR(numeric)} - Rounds *numeric* down, and returns the largest number that is less than or equal to *numeric*
- {fn PI()} - Returns a value that is closer than any other value to *pi*
- {fn RADIANS(numeric)} - Converts *numeric* from degrees to radians
- {fn RAND(numeric)}
- {fn ROUND(numeric, numeric)}
- {fn SIGN(numeric)}
- {fn SIN(numeric)} - Returns the sine of *numeric*
- {fn SQRT(numeric)} - Returns the square root of *numeric*
- {fn TAN(numeric)} - Returns the tangent of *numeric*
- {fn TRUNCATE(numeric, numeric)}

## 字符串

| 字符串                                          | 描述                                                                                            |
|----------------------------------------------|-----------------------------------------------------------------------------------------------|
| {fn CONCAT(character, character)}            | 连接字符串                                                                                         |
| {fn LOCATE(string1, string2)}                | 返回 <i>string2</i> 在 <i>string1</i> 中首次出现的位置. 如果指定了 <i>integer</i> , 则从 <i>integer</i> 为起点开搜索. |
| {fn INSERT(string1, start, length, string2)} | 把 <i>string2</i> 插入 <i>string1</i>                                                            |
| {fn LCASE(string)}                           | 返回小写                                                                                          |
| {fn LENGTH(string)}                          | 返回字符数                                                                                         |
| {fn SUBSTRING(string, offset, length)}       | 字符串截取, 从 <i>offset</i> 的位置开始截取, 长度为 <i>length</i>                                             |
| {fn UCASE(string)}                           | 返回大写                                                                                          |

已知的bug:

| 语法                                        | 描述                                                                                            |
|-------------------------------------------|-----------------------------------------------------------------------------------------------|
| {fn LOCATE(string1, string2 [, integer])} | 返回 <i>string2</i> 在 <i>string1</i> 中首次出现的位置. 如果指定了 <i>integer</i> , 则从 <i>integer</i> 为起点开搜索. |

|                    |                          |
|--------------------|--------------------------|
| {fn LTRIM(string)} | 移除 <i>string</i> 头部的空白字符 |
| {fn RTRIM(string)} | 移除 <i>string</i> 尾部的空白字符 |

Calcite 1.9.0 在函数使用位置参数的时候会抛出异常, {fn LTRIM} 和 {fn RTRIM} 在编译 SQL 语句的时候. 这个能在将来的版本中修复.

未实现的:

- {fn ASCII(string)} - 转换单个字符的字符串为 ASCII 码, 为 0 - 255 之间的整数
- {fn CHAR(string)}
- {fn DIFFERENCE(string, string)}
- {fn LEFT(string, integer)}
- {fn REPEAT(string, integer)}
- {fn REPLACE(string, string, string)}
- {fn RIGHT(string, integer)}
- {fn SOUNDEX(string)}
- {fn SPACE(integer)}

## Date/time

| 语法                                                   | 描述                                                            |
|------------------------------------------------------|---------------------------------------------------------------|
| {fn CURDATE()}                                       | 等价于 CURRENT_DATE                                              |
| {fn CURTIME()}                                       | 等价于 LOCALTIME                                                 |
| {fn NOW()}                                           | 等价于 LOCALTIMESTAMP                                            |
| {fn QUARTER(date)}                                   | 等价于 EXTRACT(QUARTER FROM date). 返回 1 和 4 之间的整数.               |
| {fn TIMESTAMPADD(timeUnit, count, timestamp)}        | 添加 <i>count</i> 个 <i>timeUnit</i> 间隔到 <i>timestamp</i>        |
| {fn TIMESTAMPDIFF(timeUnit, timestamp1, timestamp2)} | <i>timestamp1</i> 减去 <i>timestamp2</i> 结果存在 <i>timeUnit</i> 中 |

未实现的:

- {fn DAYNAME(date)}

- {fn DAYOFMONTH(date)}
- {fn DAYOFWEEK(date)}
- {fn DAYOFYEAR(date)}
- {fn HOUR(time)}
- {fn MINUTE(time)}
- {fn MONTH(date)}
- {fn MONTHNAME(date)}
- {fn SECOND(time)}
- {fn WEEK(date)}
- {fn YEAR(date)}

系统

未实现的:

- {fn DATABASE()}
- {fn IFNULL(value, value)}
- {fn USER(value, value)}
- {fn CONVERT(value, type)}

聚合函数(**aggregate functions**)

Storm SQL 当前不支持聚合函数.

窗口函数(**windowing functions**)

Storm SQL 当前不支持窗口函数

分组函数(**grouping functions**)

Storm SQL 不支持分组函数

用户定义函数(**User-defined functions**)

用户可以使用 `CREATE FUNCTION` 语句定义 user defined function(标量). 例如, 下面的语句使用 `org.apache.storm.sql.TestTools$MyPlus` 类定义了一个 `MYPLUS` 函数.

```
CREATE FUNCTION MYPLUS AS 'org.apache.storm.sql.TestTools$MyPlus'
```

Storm SQL 通过检验定义的个方法来确定函数是标量还是聚合. 如果类中定义了一个 `evaluate` 方法, Storm SQL 把这个函数作为 `标量` 对待.

标量函数的类的例子:

```
public class MyPlus {  
    public static Integer evaluate(Integer x, Integer y) {  
        return x + y;  
    }  
}
```

请注意, 用户在运行 `Storm SQL runner` 时候应当使用 `--jars` 或者 `--artifacts`, 来确保 UDFs 在 `classpath` 路径下可见.

## 外部数据源

指定外部数据源

在 `StormSQL` 中数据表现为一个外部表. 用户可以使用 `CREATE EXTERNAL TABLE` 语句指定数据源. `CREATE EXTERNAL TABLE` 的语法与 [Hive 数据定义语言](#) 定义的语法紧密相关.

```
CREATE EXTERNAL TABLE table_name field_list  
[ STORED AS  
  INPUTFORMAT input_format_classname  
  OUTPUTFORMAT output_format_classname  
]  
LOCATION location  
[ TBLPROPERTIES tbl_properties ]  
[ AS select_stmt ]
```

默认输入格式和输出格式都是 `JSON`. 我们会在将来的章节介绍 `supported formats`.

例如, 下面的语句指定了一个 `Kafka spout` 和 `sink`:

```
CREATE EXTERNAL TABLE FOO (ID INT PRIMARY KEY) LOCATION 'kafka://localhost:2181/brokers?topic=test'
```

请注意, 用户在运行 `Storm SQL runner` 时候应当使用 `--jars` 或者 `--artifacts`, 来确保 UDFs 在 `classpath` 可见.

植入外部数据源

用户通过实现 `ISqlTridentDataSource` 接口并使用 Java 的 `service loader` 机制进行注册, 以植入外部数据源. 基于表的 `URI` 的模式来选择外部数据源. 更多细节请参考 `storm-sql-kafka` 的实

现.

## 支持的格式

| 格式   | 输入格式类                                              | 输出格式类                  |
|------|----------------------------------------------------|------------------------|
| JSON | org.apache.storm.sql.runtime.serde.json.JsonScheme | org.apache.storm.sql.r |
| Avro | org.apache.storm.sql.runtime.serde.avro.AvroScheme | org.apache.storm.sql.r |
| CSV  | org.apache.storm.sql.runtime.serde.csv.CsvScheme   | org.apache.storm.sql.r |
| TSV  | org.apache.storm.sql.runtime.serde.tsv.TsvScheme   | org.apache.storm.sql.r |

## Avro

Avro 需要用户描述记录的模式(输入和输出). 模式应该在 `TBLPROPERTIES` 上描述. 输入格式需要描述给 `input.avro.schema`, 输出格式需要描述给 `output.avro.schema`. 模式字符串应当是一个转义后的 JSON, 因此 `TBLPROPERTIES` 是有效的 JSON.

示例 Schema 定义:

```
"input.avro.schema": "{\"type\": \"record\", \"name\": \"large_orders\", \"fields\" : [ {\"name\": \"
```

```
"output.avro.schema": "{\"type\": \"record\", \"name\": \"large_orders\", \"fields\" : [ {\"name\": \"
```

## CSV

使用 [Standard RFC4180 CSV Parser](#), 不需要任何其他的属性.

## TSV

默认情况下, TSV 使用 `\t` 作为分隔符, 但是用户可以通过 `input.tsv.delimiter` 或者 `output.tsv.delimiter` 设置其他的分隔符.

可支持的数据源

| 数据源     | Artifact Name                        | 位置前缀                                                |
|---------|--------------------------------------|-----------------------------------------------------|
| Socket  |                                      | socket://host:port                                  |
| Kafka   | org.apache.storm:storm-sql-kafka     | kafka://zkhost:port/broker_path?topic=topic         |
| Redis   | org.apache.storm:storm-sql-redis     | redis://:[password]@host:port/[dbIdx]               |
| MongoDB | org.apache.stormmg:storm-sql-mongodb | mongodb://[username:password@]host1[:port1][,host2[ |
| HDFS    | org.apache.storm:storm-sql-hdfs      | hdfs://host:port/path-to-file                       |

## Socket

Socket 数据源是一个内置的特性, 因此用户无需在 `--artifacts` 选项中添加任何依赖.

请注意, Socket 数据源只是用于测试: 不能保证 `exactly-once` 和 `at-least-once`.

贴士: `netcat` 是一个 Socket 便捷工具: 用户可以使用 `netcat` 连接 Socket 数据源, 既可以作为输入也可以用作输出.

## Kafka

Kafka 仅当用作输出数据源的时候需要定义下列属性:

- `producer`: 指定 Kafka Producer 配置 - 更多细节请参考 [Kafka producer configs](#).
  - `bootstrap.servers` 必须在 `producer` 中定义这个值

请注意, `storm-sql-kafka` 需要用户提供 `storm-kafka` 依赖, `storm-kafka` 又依赖于 `kafka`, `kafka-clients`. 你可以在 `--artifacts` 选项中使用下列的工作引用, 并且在需要的时候修改依赖的版本

```
org.apache.storm:storm-sql-kafka:2.0.0-SNAPSHOT,org.apache.storm:storm-kafka:2.0.0-SNAPSHOT,org.apac
```

## Redis

Redis 数据源需要设置下列属性:

- `data.type`: 用于存储的数据类型 - 仅支持 "STRING" 和 "HASH"
- `data.additional.key`: key, 当数据类型同时需要 key 和 field 时设置 (field 作为字段使用)
- `redis.timeout`: 超时时间, 毫秒 (ex. "3000")
- `use.redis.cluster`: 如果 Redis 是集群环境为 "true", 否则为 "false".

请注意, `storm-sql-redis` 需要用户提供 `storm-redis` 依赖. 你可以在 `--artifacts` 选项中使用下列的工作引用, 并且在需要的时候修改依赖的版本

```
org.apache.storm:storm-sql-redis:2.0.0-SNAPSHOT,org.apache.storm:storm-redis:2.0.0-SNAPSHOT
```

## MongoDB

MongoDB 数据源需要设置以下属性

```
{"collection.name": "storm_sql_mongo", "trident.ser.field": "serfield"}
```

- `trident.ser.field`: 存储字段 - 记录会序列化并以 BSON 存储在字段中
- `collection.name`: 集合名称

请注意, `storm-sql-mongodb` 需要用户提供 `storm-mongodb` 依赖. 你可以在 `--artifacts` 选项中使用下列的工作引用, 并且在需要的时候修改依赖的版本

```
org.apache.storm:storm-sql-mongodb:2.0.0-SNAPSHOT,org.apache.storm:storm-mongodb:2.0.0-SNAPSHOT
```

当前不支持使用保留字段存储.

## HDFS

HDFS 数据源需要设置下列属性

- `hdfs.file.path`: HDFS 文件路径
- `hdfs.file.name`: HDFS 文件名 - 参考 [SimpleFileNameFormat](#)
- `hdfs.rotation.size.kb`: HDFS FileSizeRotationPolicy 单位 KB
- `hdfs.rotation.time.seconds`: HDFS TimedRotationPolicy 单位 seconds

请注意 `hdfs.rotation.size.kb` 和 `hdfs.rotation.time.seconds` 只能采用其中一种来实现文件滚动.

还要注意 `storm-sql-hdfs` 需要用户提供 `storm-hdfs` 依赖. 你可以在 `--artifacts` 选项中使用下列的工作引用, 并且在需要的时候修改依赖的版本

```
org.apache.storm:storm-sql-hdfs:2.0.0-SNAPSHOT,org.apache.storm:storm-hdfs:2.0.0-SNAPSHOT
```

还有, 需要提供 `hdfs` 配置文件. 可以将 `core-site.xml` 和 `hdfs-site.xml` 文件放到 Storm 安装目录的 `conf` 目录下面.

# Storm SQL 内部实现

本页描述了 Storm SQL 的设计和实现.

## 概览

SQL是一个很好使用但又复杂的标准. 包括 Drill, Hive, Phoenix 和 Spark 在内的几个项目都在其 SQL 层面上投入了大量资金. StormSQL 的主要设计目标之一是利用这些项目的现有资源. StormSQL 利用[Apache Calcite](#) 来实现 SQL 标准. StormSQL 专注于将 SQL 语句编译成 Storm / Trident 拓扑, 以便它们可以在 Storm 集群中执行.

图1描述了在 StormSQL 中执行 SQL 查询的工作流程. 首先, 用户提供了一系列 SQL 语句. StormSQL 解析 SQL 语句并将其转换为 Calcite 逻辑计划. 逻辑计划由一系列 SQL 逻辑运算符组成, 描述如何执行查询而不考虑底层执行引擎. 逻辑运算符的一些示例包括 `TableScan`, `Filter`, `Projection` 和 `GroupBy`.

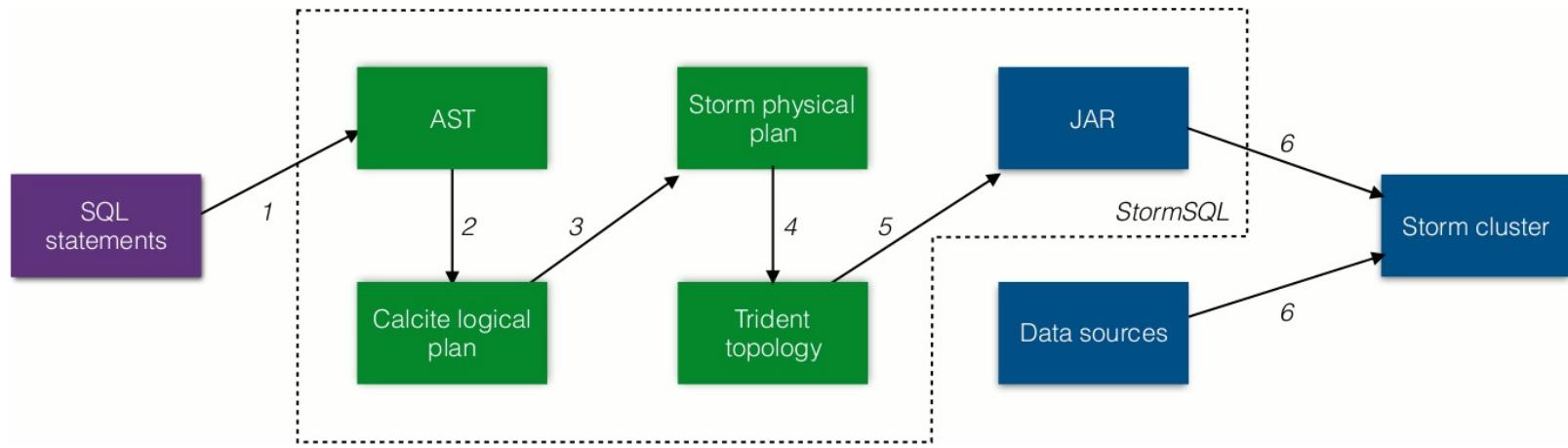


图 1: StormSQL 工作流.

下一步是将逻辑执行计划编译为物理执行计划. 物理计划由物理运算符组成, 描述如何在 `StormSQL` 中执行 SQL 查询. 物理运算符如 `Filter`, `Projection`, 和 `GroupBy` 直接映射到 Trident 拓扑中的操作. `StormSQL` 还将 SQL 语句中的表达式编译为 Java 代码块, 并将它们插入 Trident 函数, 这些函数将在运行时被编译一次并执行.

最后, `StormSQL` 将创建的 Trident 拓扑与空的打包 JAR 提交到 Storm 集群. Storm 计划并以与执行其他 Storm 拓扑相同的方式执行 Trident 拓扑.

以下代码块显示了一个示例查询, 用于过滤和投影来自 Kafka 流的结果.

```
CREATE EXTERNAL TABLE ORDERS (ID INT PRIMARY KEY, UNIT_PRICE INT, QUANTITY INT) LOCATION 'kafka://1'
```

```
CREATE EXTERNAL TABLE LARGE_ORDERS (ID INT PRIMARY KEY, TOTAL INT) LOCATION 'kafka://localhost:2181'
INSERT INTO LARGE_ORDERS SELECT ID, UNIT_PRICE * QUANTITY AS TOTAL FROM ORDERS WHERE UNIT_PRICE * Q
```

前两个 SQL 语句定义外部数据的输入和输出. 图2描述了 StormSQL 如何获取最后一个 SELECT 查询并将其编译为 Trident 拓扑的过程.

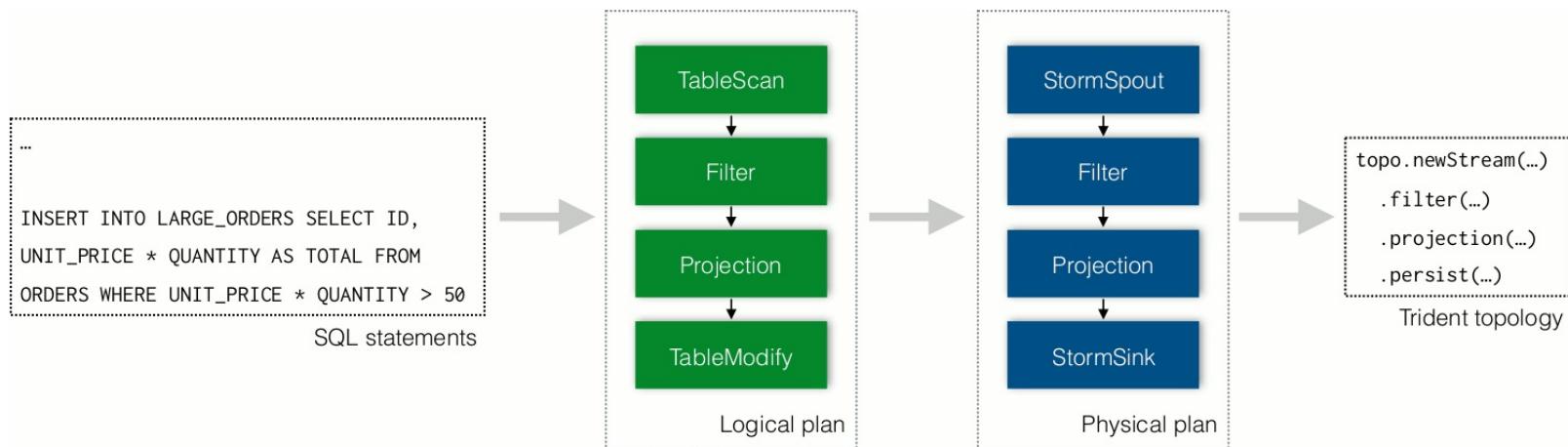


图 2: 将示例查询编译为 Trident 拓扑.

## 查询 **streaming tables** 的限制

查询代表实时数据流的表时有几个局限:

- `ORDER BY` 子句不能应用于流
- `GROUP BY` 子句中至少有一个单调字段允许 StormSQL 限制缓冲区的大小

更多细节请参考 <http://calcite.apache.org/docs/stream.html>.

## 依赖

Storm 除了 `EXTERNAL TABLE` 使用的数据源 JAR 外,还要注意 Storm SQL 的必须的依赖. 您可以使用 `storm sql` 的 `--jars` 或 `--artifacts`,以便数据源 JAR 可以包含在 Storm SQL Runner 中,也可以包含在 Trident 拓扑运行时 `classpath` 中. (如果您的数据源 JAR 在 Maven 存储库中可用,则使用 `--artifacts`,因为它处理传递依赖关系)

请参考 [Storm SQL集成](#).

# Flux

一个框架，为了让创建和部署Apache Storm“流”计算遇上更方便快捷  
定义

**flux** |fləks| 名词

1. 流动或者流出的这个动作过程 这个动作或者流出的过程
2. 持续不断的改变
3. 在物理中，表明液体、辐射能或者颗粒在指定区域内的流速
4. 一个混合了固体用来降低其熔点的物质

## 基本原理

当配置很难被编程的时候会发生糟糕的事情。没有人应因为需要修改配置而重新编译或者重新打包一个应用。

## 相关

Flux是一个用来让规定和部署Apache Storm拓扑不那么费劲的框架和一系列工具。

你是否发现你总是重复这样的模式？：

```
public static void main(String[] args) throws Exception {
    // 返回的逻辑值用来判断我们是否在本地上运行
    // 创建必要的配置选项...
    boolean runLocal = shouldRunLocal();
    if(runLocal){
        LocalCluster cluster = new LocalCluster();
        cluster.submitTopology(name, conf, topology);
    } else {
        StormSubmitter.submitTopology(name, conf, topology);
    }
}
```

像这样的操作会不会更容易一些呢：

```
storm jar mytopology.jar org.apache.storm.flux.Flux --local config.yaml
```

或者：

```
storm jar mytopology.jar org.apache.storm.flux.Flux --remote config.yaml
```

另一个比较经常提及的麻烦点在于由于写拓扑图常常是和Java代码紧密结合的，所以任何对它的修改都需要重新编译和重新打包拓扑的jar文件。Flux的目标是允许你将你所有的Storm组件打包在一个单独的jar文件中，然后使用另一个文本文件来规定你的拓扑的布局和配置。通过这样的方式，缓解这一个麻烦。

## 特点

- 安装和部署Storm拓扑（包括Storm core和Microbatch API）简单，而不是用内嵌的配置方法在你的拓扑代码中安装和部署。
- 支持已有的拓扑代码（如下可见）
- 通过使用灵活的YAML DSL定义Storm Core API（Spouts/Bolts）。
- YAML DSL支持大多数的Storm组件 (storm-kafka, storm-hdfs, storm-hbase, 等等.)
- 对多语言的组件有便捷的支持
- 为了更简便地在配置/环境间转换，使用了外部属性的置换/过滤（类似于Maven风格的\${variable.name}置换）。

## 用法

为了使用Flux，把它添加到依赖包中，然后把你所有的Storm组件打包成一个很大的jar文件，再然后创建一个YAML文件来定义你的拓扑（下面有YAML配置选项）。

### 通过源码来构建

使用Flux最简单的方法就是将它作为Maven依赖包添加到项目中，如下面描述的那样。

如果你要从源代码中创建Flux并进行单元/集成的测试，你需要在你的系统上按照如下操作来安装：

- Python 2.6.x or later
- Node.js 0.10.x or later

创建能够使用单元测试的命令：

```
mvn clean install
```

创建不能够使用单元测试的命令：

如果你希望在不安装Python和Node.js的情况下构建Flux，你可以跳过这个单元测试：

```
mvn clean install -DskipTests=true
```

需要注意，如果你打算使用**Flux**来给远程的簇部署拓扑，你仍然需要安装**Python**，因为**Apache Storm**要求这么做。

创建能够使用集成测试的命令：

```
mvn clean install -DskipIntegration=false
```

和**Maven**一起打包

为了保证**Flux**能对你的**Storm**组件有效，你需要把**Flux**当做依赖包添加，这样才能让它包含在**Storm**的拓扑jar文件中。这个可以通过**Maven shade**插件（推荐）或者**Maven assembly**插件（不推荐）来完成。

## Flux Maven依赖包

**Flux**现在的版本可以在以下的合作方的**Maven**中心获得：

```
<dependency> <groupId>org.apache.storm</groupId> <artifactId>flux-core</artifactId> <version>${s...
```

使用**shell**的**spouts**和**bolt**要求附加的**Flux Wrappers**库：

```
<dependency> <groupId>org.apache.storm</groupId> <artifactId>flux-wrappers</artifactId> <version>...
```

创建一个允许使用**Flux**的拓扑**JAR**文件

下述的例子阐述了如何配合**Maven shade**插件使用**Flux**：

```
<!-- 在shaded jar文件中包含FLux和用户依赖包 -->
<dependencies>
    <!-- Flux include -->
    <dependency>
        <groupId>org.apache.storm</groupId>
        <artifactId>flux-core</artifactId>
        <version>${storm.version}</version>
    </dependency>
    <!-- Flux Wrappers include -->
    <dependency>
        <groupId>org.apache.storm</groupId>
        <artifactId>flux-wrappers</artifactId>
        <version>${storm.version}</version>
    </dependency>

    <!-- 在这里添加用户依赖包... -->

</dependencies>
<!-- 创建一个包括所有依赖包的大大的jar文件 -->
<build>
```

```

<plugins>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-shade-plugin</artifactId>
    <version>1.4</version>
    <configuration>
      <createDependencyReducedPom>true</createDependencyReducedPom>
    </configuration>
    <executions>
      <execution>
        <phase>package</phase>
        <goals>
          <goal>shade</goal>
        </goals>
        <configuration>
          <transformers>
            <transformer
              implementation="org.apache.maven.plugins.shade.resource.Service
              transformer"
              implementation="org.apache.maven.plugins.shade.resource.Manifes
              tResource"
              mainClass="org.apache.storm.flux.Flux"/>
            </transformer>
          </transformers>
        </configuration>
      </execution>
    </executions>
  </plugin>
</plugins>
</build>

```



## 部署和运行 Flux 拓扑

一旦你的拓扑组件和 Flux 的依赖包一起打包后，你就可以通过使用 `storm jar` 命令在本地或者远端运行不同的拓扑。比如说，如果你的大大的 jar 文件命名为 `myTopology-0.1.0-SNAPSHOT.jar`，你可以使用以下的命令在本地运行它：

```
storm jar myTopology-0.1.0-SNAPSHOT.jar org.apache.storm.flux.Flux --local my_config.yaml
```

## 命令行的参数选项

```
usage: storm jar <my_topology_uber_jar.jar> org.apache.storm.flux.Flux
[options] <topology-config.yaml>
```

<code>-d, --dry-run</code>	不运行/部署这个拓扑，仅仅是构建、验证和打印这个拓扑的相关信息。
<code>-e, --env-filter</code>	执行环境变量的替换。以形式为 `\${ENV-[NAME]}` 定义的替换关键字将会替换 `
<code>-f, --filter &lt;file&gt;</code>	执行属性替换。使用一个指定的文件作为属性的源，然后形式为 \${[property name]}
<code>-i, --inactive</code>	部署但是不激活这个拓扑。
<code>-l, --local</code>	在 local 的模式下运行拓扑。
<code>-n, --no-splash</code>	抑制版权页的输出。
<code>-q, --no-detail</code>	抑制拓扑详情的输出。
<code>-r, --remote</code>	将拓扑部署到远端的簇。
<code>-R, --resource</code>	使用提供的路径来作为 classpath 的源文件以代替文件。

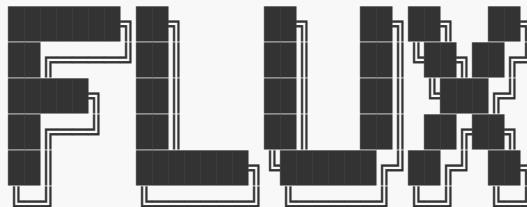
-s,--sleep <ms> 当在本地运行时，在killing一个拓扑和关闭本地簇之前需要sleep的时间（以ms为单位）  
-z,--zookeeper <host:port> 当以local模式运行时，使用ZooKeeper的特定<host:port>而不是同进程的ZooKeeper

注意： Flux为了避免在使用到 `storm` 命令时产生命令行开关冲突，所以允许任何其他的命令行开关来表达 `storm` 这一命令。

举例来说，你可以使用 `storm` 的命令开关 `-c` 来覆盖拓扑配置性能。下述举例的命令就可以运行 Flux 并且覆盖 `nimbus.seeds` 这一配置：

```
storm jar myTopology-0.1.0-SNAPSHOT.jar org.apache.storm.flux.Flux --remote my_config.yaml -c 'nimbus.seeds' '127.0.0.1:2379'
```

输出例子



```
+-- Apache Storm      +-+
+-+ data FLow User eXperience +-+
Version: 0.3.0
Parsing file: /Users/hsimpson/Projects/donut_domination/storm/shell_test.yaml
----- TOPOLOGY DETAILS -----
Name: shell-topology
----- SPOUTS -----
sentence-spout[1](org.apache.storm.flux.wrappers.spouts.FluxShellSpout)
----- BOLTS -----
splitsentence[1](org.apache.storm.flux.wrappers.bolts.FluxShellBolt)
log[1](org.apache.storm.flux.wrappers.bolts.LogInfoBolt)
count[1](org.apache.storm.testing.TestWordCounter)
----- STREAMS -----
sentence-spout --SHUFFLE--> splitsentence
splitsentence --FIELDS--> count
count --SHUFFLE--> log
-----
Submitting topology: 'shell-topology' to remote cluster...
```

## YAML 配置

Flux 拓扑在 YAML 文件中被顶迎来描述一个拓扑。一个 Flux 拓扑定义由以下几项组成：

1. 一个拓扑的名字
2. 一个拓扑组件的列表（被命名为 Java 对象，用以可以在环境中可以调用）
3. 第三选项或者第四选项（一个 DSL 拓扑定义）：
  - 一个 spouts 的列表，每一个项通过一个唯一的 ID 被识别

- 一个**bolts**的列表，每一个项通过一个唯一的ID被识别
- 一个“**stream**”对象的列表，用来表示**spouts**和**bolts**之间的流的元组。

4. 第三选项或者第四选项 (一个可以创建 `org.apache.storm.generated.StormTopology` 实例的 JVM类) :
- 一个 `topologySource` 定义。

举个例子，这里有一个使用YAML DSL的简单wordcount拓扑：

```

name: "yaml-topology"
config:
  topology.workers: 1

# spout定义
spouts:
  - id: "spout-1"
    className: "org.apache.storm.testing.TestWordSpout"
    parallelism: 1

# bolt定义
bolts:
  - id: "bolt-1"
    className: "org.apache.storm.testing.TestWordCounter"
    parallelism: 1
  - id: "bolt-2"
    className: "org.apache.storm.flux.wrappers.bolts.LogInfoBolt"
    parallelism: 1

# stream定义
streams:
  - name: "spout-1 --> bolt-1" #name暂时未用上 (可以在logging,UI等中作为placeholder)
    from: "spout-1"
    to: "bolt-1"
    grouping:
      type: FIELDS
      args: ["word"]

  - name: "bolt-1 --> bolt2"
    from: "bolt-1"
    to: "bolt-2"
    grouping:
      type: SHUFFLE

```

## 属性替换/过滤

对于开发者而言，想要简单地转换配置是常有的事，例如在开发环境和生产环境中转换部署。这可以通过使用分开的YAML配置文件来实现，但是这个方法会导致配置文件中多出一些多余的重复内容，尤其是在一些Storm拓扑没有改变但是配置设置例如主机名，端口和并

行性参数改变了的情况。

对于这种情况，**Flux**提供了属性过滤（properties filtering）方法允许你给一个 `.properties` 文件赋两个具体的值，并且让他们在 `.yaml` 文件被解析前被替代。

为了实现属性过滤功能，使用 `--filter` 命令行选项，并且具体制定一个 `.properties` 文件。举个例子，如果你像这样调用**flux**:

```
storm jar myTopology-0.1.0-SNAPSHOT.jar org.apache.storm.flux.Flux --local my_config.yaml --filter
```

并且 `dev.properties` 内容如下：

```
kafka.zookeeper.hosts: localhost:2181
```

你在这之后就可以通过你 `.yaml` 文件中的属性关键字，使用 `{}$` 语法来引用它：

```
- id: "zkHosts"
  className: "org.apache.storm.kafka.ZkHosts"
  constructorArgs:
    - "${kafka.zookeeper.hosts}"
```

在这个例子中，**Flux**可以在YAML内容被解析前使用 `localhost:2181` 来替换  `${kafka.zookeeper.hosts}` 。

## 环境变量替换/过滤 Environment Variable Substitution/Filtering

**Flux**同样也允许环境变量替换。举个例子，如果名为`ZK_HOSTS` 的环境变量名被定义了，你可以通过以下的语法在**Flux**的YAML文件中引用它：

```
 ${ENV-ZK_HOSTS}
```

## 组件

组件从本质来说是对象实例，用来在对**spouts**和**bolts**的配置选项中获取。如果你对**Spring**框架很熟悉，这里的组件大概就类比于**Spring**中的**beans**

每一个组件都是可被识别的，至少是可以通过一个唯一的标识符（字符串）和一个类名（字符串）。举个例子，以下的例子将会创建一个 `org.apache.storm.kafka.StringScheme` 类的实例作为关键字 `"stringScheme"` 的引用。这里我们假设这个类 `org.apache.storm.kafka.StringScheme` 有一个默认的构造函数。

```
components:
- id: "stringScheme"
  className: "org.apache.storm.kafka.StringScheme"
```

## 构造函数参数，引用，属性和配置方法

### 构造函数参数

为了给一个类的构造函数添加参数，我们添加 `constructorArgs` 这个元素给组件。

`constructorArgs` 是一个列表，其元素是对象。这个列表会被传递给类的构造函数们。以下的这个例子通过调用一个把单个字符串作为参数的构造函数来创建一个对象：

```
- id: "zkHosts"
  className: "org.apache.storm.kafka.ZkHosts"
  constructorArgs:
    - "localhost:2181"
```

### 引用

每一个组件实例都通过一个唯一的 `id` 可被其他组件重复使用。为了引用一个已存在的组件，你需要在使用 `ref` 这个标签的时候指明这个组件的 `id`。

在以下的例子中，一个名为的组件被创建，之后将被作为另一个组件的构造函数的参数被引用：

```
components:
- id: "stringScheme"
  className: "org.apache.storm.kafka.StringScheme"

- id: "stringMultiScheme"
  className: "org.apache.storm.spout.SchemeAsMultiScheme"
  constructorArgs:
    - ref: "stringScheme" # component with id "stringScheme" must be declared above.
```

注意：引用只能在对象被声明后使用。

### 属性

除去允许在调用构造函数的时候传进不同的参数，Flux同样允许在配置组件的时候使用被声明为 `public` 的类似JavaBean的`setter`方法和域：

```
- id: "spoutConfig"
  className: "org.apache.storm.kafka.SpoutConfig"
  constructorArgs:
    # brokerHosts
    - ref: "zkHosts"
```

```

# topic
- "myKafkaTopic"
# zkRoot
- "/kafkaSpout"
# id
- "myId"
properties:
- name: "ignoreZkOffsets"
  value: true
- name: "scheme"
  ref: "stringMultiScheme"

```

在上述的例子中，如果声明了 `properties`，`Flux`将会在 `SpoutConfig` 中找一个名字为 `setIgnoreZkOffsets(boolean b)` 的函数并试图调用它。如果这样的一个 `setter` 函数没有找到，`Flux` 就会找一个公有的叫 `ignoreZkOffsets` 的实例变量并且将它进行设置。

引用也可能被作为属性值来使用。

## 配置方法

从概念上来说，配置方法可能类似于属性和构造函数的参数——他们允许一个对象在创建后可以调用任意的方法。对于有一些类，它们没有暴露 `JavaBean` 的方法或者没有能够将整个对象都配置好的构造函数，配置方法对这种类就十分有用。一些常见的例子包括了哪些使用构造器模式来配置/整合的类。

接下来的 `YAML` 例子创建了一个 `bolt` 并且通过几个方法进行了配置：

```

bolts:
- id: "bolt-1"
  className: "org.apache.storm.flux.test.TestBolt"
  parallelism: 1
  configMethods:
    - name: "withFoo"
      args:
        - "foo"
    - name: "withBar"
      args:
        - "bar"
    - name: "withFooBar"
      args:
        - "foo"
        - "bar"

```

对应方法的标识如下：

```

public void withFoo(String foo);
public void withBar(String bar);
public void withFooBar(String foo, String bar);

```

传递给配置方法的参数和构造函数中的参数作用一样，并且也支持引用。

## 在沟早期的参数，引用，属性和配制方法中使用Java的 `enums in Constructor Arguments, References, Properties and Configuration Methods`

你可以在Flux YAML文件中轻松通过引用 `enum` 的名字将其值作为参数。

比如，[Storm's HDFS 模块](#) 包含了以下 `enum` 的定义（为了简洁而简化过）：

```
public static enum Units {  
    KB, MB, GB, TB  
}
```

`org.apache.storm.hdfs.bolt.rotation.FileSizeRotationPolicy` 这个类有如下的构造器：

```
public FileSizeRotationPolicy(float count, Units units)
```

以下的Flux `component` 定义可以被用来调用这个构造器：

```
- id: "rotationPolicy"  
  className: "org.apache.storm.hdfs.bolt.rotation.FileSizeRotationPolicy"  
  constructorArgs:  
    - 5.0  
    - MB
```

上述的定义和下面的Java代码从功能上来说是一样的：

```
// rotate files when they reach 5MB  
FileRotationPolicy rotationPolicy = new FileSizeRotationPolicy(5.0f, Units.MB);
```

## 拓扑配置

`config` 这个区段仅仅是Storm拓扑配置参数的一个图，将会作为 `org.apache.storm.Config` 类的实例传给 `org.apache.storm.StormSubmitter`：

```
config:  
  topology.workers: 4  
  topology.max.spout.pending: 1000  
  topology.message.timeout.secs: 30
```

## 已经存在的拓扑

如果你有已经存在的Storm拓扑，你仍然可以用Flux来部署/运行/测试它们。这个特点允许你按照已有的拓扑类来改变Flux构造参数，引用，属性和拓扑配置声明。

使用已有拓扑类最简单的方法就是通过下面的方法定义一个名为 `getTopology()` 的实例方法：

```
public StormTopology getTopology(Map<String, Object> config)
```

或者：

```
public StormTopology getTopology(Config config)
```

你接下来就可以使用YAML来部署你的拓扑：

```
name: "existing-topology"
topologySource:
  className: "org.apache.storm.flux.test.SimpleTopology"
```

如果你想用来作为拓扑源的类有一个不同的方法名（比如不叫），那么你可以把它重写：

```
name: "existing-topology"
topologySource:
  className: "org.apache.storm.flux.test.SimpleTopology"
  methodName: "getTopologyWithDifferentMethodName"
```

注意：这个指定的方法必须接受一个单一的类型是 `java.util.Map<String, Object>` 或者 `org.apache.storm.Config` 的类，然后返回一个 `org.apache.storm.generated.StormTopology` 对象。

## YAML DSL

### Spouts 和 Bolts

Spout和Bolts是在YAML配置中各自的区域中被配置。Spout和Bolt的定义是 `组件 (component)` 定义的拓展。在组件的基础上添加了 `并行度 (parallelism)` 参数，用于当一个拓扑被部署的时候设置组件的并行度。

因为spout和bolt定义继承了 `组件 (component)`，所以它们也支持构造函数参数，引用，属性。Because spout and bolt definitions extend they support constructor arguments, references, and properties as well.

Shell spout的例子：

```
spouts:
- id: "sentence-spout"
  className: "org.apache.storm.flux.wrappers.spouts.FluxShellSpout"
  # shell spout constructor takes 2 arguments: String[], String[]
  constructorArgs:
    # command line
    - ["node", "randomsentence.js"]
```

```
# output fields
- ["word"]
parallelism: 1
```

## Kafka spout的例子：

```
components:
- id: "stringScheme"
  className: "org.apache.storm.kafka.StringScheme"

- id: "stringMultiScheme"
  className: "org.apache.storm.spout.SchemeAsMultiScheme"
  constructorArgs:
    - ref: "stringScheme"

- id: "zkHosts"
  className: "org.apache.storm.kafka.ZkHosts"
  constructorArgs:
    - "localhost:2181"

# 可选的kafka配置
# - id: "kafkaConfig"
#   className: "org.apache.storm.kafka.KafkaConfig"
#   constructorArgs:
#     # brokerHosts
#     - ref: "zkHosts"
#     # topic
#     - "myKafkaTopic"
#     # clientId (optional)
#     - "myKafkaClientId"

- id: "spoutConfig"
  className: "org.apache.storm.kafka.SpoutConfig"
  constructorArgs:
    # brokerHosts
    - ref: "zkHosts"
    # topic
    - "myKafkaTopic"
    # zkRoot
    - "/kafkaSpout"
    # id
    - "myId"
  properties:
    - name: "ignoreZkOffsets"
      value: true
    - name: "scheme"
      ref: "stringMultiScheme"

config:
  topology.workers: 1

# spout definitions
spouts:
- id: "kafka-spout"
  className: "org.apache.storm.kafka.KafkaSpout"
  constructorArgs:
```

```
- ref: "spoutConfig"
```

Bolt 例子：

```
# bolt definitions
bolts:
- id: "splitsentence"
  className: "org.apache.storm.flux.wrappers.bolts.FluxShellBolt"
  constructorArgs:
    # command line
    - ["python", "splitsentence.py"]
  # output fields
  - ["word"]
  parallelism: 1
  # ...
- id: "log"
  className: "org.apache.storm.flux.wrappers.bolts.LogInfoBolt"
  parallelism: 1
  # ...
- id: "count"
  className: "org.apache.storm.testing.TestWordCounter"
  parallelism: 1
  # ...
```

## Streams and Stream Groupings

Flux中的“流”被表示为一列在Spouts和Bolts之间的“连接”（如图的边，数据流等），在连接定义的同时有一个关联的“分组”定义。

一个“流”定义有如下的属性：

**name**: 一个“连接”的名字（可选的，当下不会马上使用）

**from**: 作为源头的Spout或者Bolt的 **id**（类似于出版商）

**to**: 作为目的地的Spout或者Bolt的 **id**（类似于订阅者）

**grouping**: 为了“流”而产生的“流分组”定义

一个“分组”定义有以下的属性：

**type**: 分组的类型。下列值中任选一个

**ALL**、**CUSTOM**、**DIRECT**、**SHUFFLE**、**LOCAL\_OR\_SHUFFLE**、**FIELDS**、**GLOBAL**、或者 **NONE**。

**streamId**: Storm “流”的ID（可选的，如果没有指定则会使用默认流）

`args`: 当 `type` 的值为 `FIELDS` 时，一系列域的名字。

`customClass` 当 `type` 的值为 `CUSTOM` 时，自定义的“分组”类实例的定义

如下例的 `流 (streams)` 的定义案例建立起了一个如下的线路拓扑：

```
kafka-spout --> splitsentence --> count --> log
```

```
# 流 (stream) 定义
# “流”的定义定了在spouts和bolts之间的“连接”。
# 注意这样的“连接”可能是循环的
# 自定义的“流分组”也被支持
```

```
streams:
```

```
- name: "kafka --> split" # name isn't used (placeholder for logging, UI, etc.)
  from: "kafka-spout"
  to: "splitsentence"
  grouping:
    type: SHUFFLE

- name: "split --> count"
  from: "splitsentence"
  to: "count"
  grouping:
    type: FIELDS
    args: ["word"]

- name: "count --> log"
  from: "count"
  to: "log"
  grouping:
    type: SHUFFLE
```

## 自定义“流分组”

自定义的流分组是通过设置分组的类型为 `CUSTOM` 并且定义一个 `customClass` 参数，该参数告诉Flux如何实例化一个自定义类。这个 `customClass` 定义继承自 `组件 (component)`，所以它也支持构造函数参数，引用和属性。

如下的例子创建了一个“流”，并且使用了一个类型为 `org.apache.storm.testing.NGrouping` 的自定义“流分组”类。

```
- name: "bolt-1 --> bolt2"
  from: "bolt-1"
  to: "bolt-2"
  grouping:
    type: CUSTOM
    customClass:
      className: "org.apache.storm.testing.NGrouping"
      constructorArgs:
```

## “包含”和“重写”

FLux允许包含其他YAML文件的内容，并且把它们当做在一个文件中定义的一样。可以包含文件或者路径源文件。

“包含”是通过一系列的maps来指定的：

```
includes:
  - resource: false
    file: "src/test/resources/configs/shell_test.yaml"
    override: false
```

如果 `resource` 的值为 `true`，“包含”将会从 `file` 这个属性值中来加载路径源文件，否则它会被当做是普通的文件。

`override` 属性控制着“包含”要如何影响定义在当前文件中的值。如果 `override` 的值是 `true`，那么 `file` 值将会替代现在的文件被解析。如果 `override` 的值是 `false`，那么当前文件正在解析的值会有优先权，并且解析器会拒绝将它们替换掉。

注意：“包含”现今不是循环的，包含文件中的包含将会被忽视。

## 基本的Word Count例子

这个例子使用了在JavaScript中实现的spout，Python中实现的bolt，和另一个在Java中实现的bolt。

拓扑 YAML 配置：

```
---
name: "shell-topology"
config:
  topology.workers: 1

# spout 定义
spouts:
  - id: "sentence-spout"
    className: "org.apache.storm.flux.wrappers.spouts.FluxShellSpout"
    # shell spout constructor takes 2 arguments: String[], String[]
    constructorArgs:
      # command line
      - ["node", "randomsentence.js"]
      # output fields
      - ["word"]
parallelism: 1
```

```

# bolt 定义
bolts:
  - id: "splitsentence"
    className: "org.apache.storm.flux.wrappers.bolts.FluxShellBolt"
    constructorArgs:
      # command line
      - ["python", "splitsentence.py"]
    # output fields
    - ["word"]
  parallelism: 1

  - id: "log"
    className: "org.apache.storm.flux.wrappers.bolts.LogInfoBolt"
    parallelism: 1

  - id: "count"
    className: "org.apache.storm.testing.TestWordCounter"
    parallelism: 1

#stream 定义
# “流”定义定义了在spouts和bolts之间的连接
# 注意“连接”可能是循环的
# 自定义“流分组”也是被支持的

streams:
  - name: "spout --> split" # name没有被使用 ( 是logging, UI等中的占位符)
    from: "sentence-spout"
    to: "splitsentence"
    grouping:
      type: SHUFFLE

  - name: "split --> count"
    from: "splitsentence"
    to: "count"
    grouping:
      type: FIELDS
      args: ["word"]

  - name: "count --> log"
    from: "count"
    to: "log"
    grouping:
      type: SHUFFLE

```

## Micro-Batching (Trident) API 支持

虽然目前Flux DSL只支持核心Storm API (the COrer Storm API)，但是对Storm的micro-batching API的支持正在计划中。

为了和Trident拓扑一起使用Flux，在你的YAML配置中定义一个拓扑的getter方法和引用：

```
```yaml
name: "my-trident-topology"
```

```
config: topology.workers: 1
```

topologySource: className: "org.apache.storm.flux.test.TridentTopologySource" # FLux将  
会寻找 "getTopology"方法, 这个会用来重写之前那个 methodName:  
"getTopologyWithDifferentMethodName" `

# Storm 安装和部署

# 设置Storm集群

本页概述了启动和运行Storm集群的步骤,如果您使用AWS,您应该查看[storm-部署](#)项目.[storm-部署](#) 在EC2上完全自动化配置和安装Storm集群. 它还为您设置Ganglia,以便您可以监视CPU,磁盘和网络使用情况.

如果您在运行Strom集群时遇到困难,请首先在 [Troubleshooting](#) 页寻求解决. 再者, 查看或者发送邮件列表.

下面是部署Storm集群的步骤总结:

1. 设置 Zookeeper 集群.
2. 设置Nimbus和worker 节点的安装环境
3. 在集群节点下载解压 Storm
4. 在storm.yaml中设置必要的配置
5. 在指导下使用 storm 脚本来启动相应的守护进程. (备注:master包括nimbus和UI进程,slave包括supervisor和logviewer)

## 设置Zookeeper集群

Storm 使用 Zookeeper 协调管理集群. Zookeeper 并不是 用于消息传递, 所以 Storm 对 Zookeeper造成的负载压力非常低. 单节点Zookeeper集群在大多数情况下应该是足够的,但是如果您想要故障转移或部署大型Storm集群,则可能需要较大的Zookeeper集群. 部署Zookeeper的说明是[这里](#).

关于Zookeeper部署的几点注意事项:

- 1.在监督下运行Zookeeper至关重要,因为Zookeeper是故障快速的,如果遇到任何错误的情况都将退出进程. 有关详细信息,请参阅[这里](#) .
- 2.建立一个cron定时任务来压缩Zookeeper的数据和事务日志至关重要. Zookeeper守护进程本身不会这样做,如果没有设置cron,Zookeeper将很快耗尽磁盘空间. 有关详细信息,请参阅[这里](#).

## Nimbus 和 worker 节点的安装环境

接下来你需要准备Nimbus 和 worker 节点的安装环境:

1. Java 7 (备注:建议使用Java 8,毕竟很多的项目开发环境已经迁移到8以上)
2. Python 2.6.6

这些依赖版本是Storm已经测试过的. Storm 在不同的Java 或Python版本上也许会存在问题.

## 下载解压 Storm

接下来,下载一个Storm版本,并解压zip文件到Nimbus和每个worker机器上的某个目录下.  
Storm版本可以从[这里](#)下载.

在**storm.yaml**中设置必要的配置

Storm 发布包中在目录`conf/storm.yaml` 下包含一个默认的配置文件. 你可以[在这里](#)查看默认值. `storm.yaml` 中的存在的配置项会覆盖掉 `defaults.yaml`中相应的配置项. 下面一些配置是集群运行时所必要的:

1) **storm.zookeeper.servers**: 这是一个Storm集群所依赖 Zookeeper 集群的hosts列表. 类似于:

```
storm.zookeeper.servers:  
- "111.222.333.444"  
- "555.666.777.888"
```

如果配置的Zookeeper集群不是默认的端口, 你应该设置 **storm.zookeeper.port** 选项.

2) **storm.local.dir**: Nimbus 和 Supervisor 守护进程需要配置一个本地目录来存储少量状态信息(例如jars包,配置文件等等). 您应该在每个机器上创建该目录,给予适当的权限,然后使用此配置填写目录位置. 例如:

```
storm.local.dir: "/mnt/storm"
```

如果您在windows下运行Strom,应该如下: `yaml storm.local.dir: "C:\\\\storm-local"` 如果您使用相对路径,那么路径是相对于(STORM\_HOME). 您也可以使用默认值 `$STORM_HOME/storm-local`

3) **nimbus.seeds**: worker节点需要知道哪些机器是主机的候选者,以便下载 topology jar和 confs(nimbus.host 在1.0之后已经废弃,这里实现了HA). 例如:

```
nimbus.seeds: ["111.222.333.444"]
```

鼓励您填写\*机器的FQDN \*(Fully Qualified Domain Name,全域名)列表. 如果要设置Nimbus

HA,则必须解决运行nimbus的所有机器的FQDN.当您只想设置“伪分布式”集群时您可能希望将其保留为默认值,仍然鼓励您填写FQDN.

4) **supervisor.slots.ports**: 对于每个worker节点,您可以使用此配置设置在该计算机上运行的worker数量. 每个worker使用单个端口接收消息,并且此设置定义哪些端口打开以供使用. 如果您在此定义五个端口,那么Storm将分配最多五个worker在本机上运行. 如果您定义了三个端口,Storm将只能运行三个worker. 默认情况下,此设置被配置为在端口6700,6701,6702和6703上运行4个worker:

```
supervisor.slots.ports:
```

- 6700
- 6701
- 6702
- 6703

## 监控 **Supervisors** 健康状态

Storm提供了一种机制,管理员可以通过该机制配置supervisor进程定期运行管理员提供的脚本,以确定节点是否健康. 管理员可以通过执行位于`storm.health.check.dir`中的脚本来确定supervisor节点是否处于健康状态. 如果脚本检测到节点处于不正常状态,则必须标准流输出ERROR开头的信息. supervisor将定期运行健康检查目录中的脚本并检查输出. 如果脚本的输出包含字符串ERROR,如上所述,superviosr进程将关闭所有worker并退出.

如果supervisor 已经有监控脚本,通过执行 "/bin/storm node-health-check"可以确定supervisor 是否可以启动 或 supervisor 节点是否健康

健康检查目录的地址通过以下配置项设置:

```
storm.health.check.dir: "healthchecks"
```

脚本必须有执行权限. 健康检查脚本可以设置超时失败,在时间段内脚本未执行完毕则被标记为失败,脚本运行超时时间设置:

```
storm.health.check.timeout.ms: 5000
```

## 配置外部**libs**和环境变量 (可选)

490/5000 如果需要外部库或自定义插件的支持,可以将这些jar放在extlib/ 和 extlib-daemon/ 目录中. 请注意,extlib-daemon/ 目录存储仅由守护进程

(Nimbus, Supervisor, DRPC, UI, Logviewer) 所调用加载的jar,例如HDFS和自定义调度库. 因此,用户可以配置两个环境变量STORM\_EXT\_CLASSPATH和STORM\_EXT\_CLASSPATH\_DAEMON,以便包含外部classpath和仅限守护进程调用的外部classpath.

在指导下使用 "**storm**" 脚本启动相应的守护进程

最后一步是启动所有的Storm守护进程. 在指导下运行这些守护进程是至关重要的. Storm是一个**fail-fast**系统,这意味着每当遇到意外错误时,进程将停止. Storm的设计使得它可以在任何时候安全地停止,并在进程重新启动时正确恢复. 这就是为什么Storm不会在进程中保持状态 - 如果Nimbus或者Supervisors重新启动,运行的topologies不会受到影响. 以下是运行Storm守护进程的方法:

1. **Nimbus**: 在主节点(Nimbus)运行命令行 "bin/storm nimbus".
2. **Supervisor**: 在每个supervisor节点运行命令行 "bin/storm supervisor". supervisor守护程序负责启动和停止该机器上的worker进程.
3. **UI**: 通过运行命令"bin/storm ui",运行Storm UI (一般选择在Nimbus节点启动一个,可以从浏览器访问群集和topologies的诊断信息). 可以通过浏览网页浏览器访问<http://{ui host}:8080>.

正如你所看到的,运行守护进程非常简单. 守护进程将日志输出到您解压缩Storm时的 logs/ 目录.

# 本地模式

本地模式是一种在本地进程中模拟Storm集群的工作模式,对开发和测试 topologies (拓扑) 非常有用.本地模式运行 topologies (拓扑) 和在[集群上](#)运行 topologies 一样。

创建一个进程内的集群, 只需要使用 LocalCluster 类. 例如:

```
import org.apache.storm.LocalCluster;  
  
LocalCluster cluster = new LocalCluster();
```

然后,您可以使用 `LocalCluster` 对象的 `submitTopology` 方法提交topologies (拓扑), 和 `StormSubmitter`中的一些方法相似, `submitTopology` 以 拓扑名称, topology configuration, 和 topology 对象作为参数. 你可以使用 `killTopology` 方法 kill a topology, `killTopology` 方法接受 topology name 为参数 使用以下语句关闭本地模式集群:

```
cluster.shutdown();
```

本地模式的常用配置

您可以在[这里](#)看到所有的配置 config列表.

1. **Config.TOPOLOGY\_MAX\_TASK\_PARALLELISM**: 该配置项设置了单个组件 (bolt/spout) 的线程数上限。生产环境上的 topology 往往含有很高的并行度（数百个线程）, 导致在本地模式下测试 topology (拓扑) 时会有较大的负载。这个配置项可以让你很容易地控制并行度。
2. **Config.TOPOLOGY\_DEBUG**: 此配置项设置为 true 时, spout 或者 bolt 每一次发送 tuple 的时候, Storm都会打印日志。这个功能对于调试很有用。

# 疑难解答

本页列出了用户使用Storm时遇到的问题及其解决方案.

## worker进程在启动时崩溃,没有堆栈跟踪(stack trace)

可能的现象:

- Topologies 在单个节点工作正常, 但是多个节点时崩溃

解决方案:

- 可能部分节点网络配置错误, 其中节点无法根据其主机名定位其他节点. 当无法解析主机时, ZeroMQ该进程有时会崩溃. 有两个解决方案:
  - 在 /etc/hosts 文件中配置主机名与IP地址的映射表
  - 设置内部DNS, 以便节点可以根据主机名相互定位.

节点无法相互通信

可能的现象:

- 每个 spout tuple 都失败了
- 处理不起作用

解决方案:

- Storm不适用于ipv6. 你可以通过添加强制ipv4 `-Djava.net.preferIPv4Stack=true` 到 supervisor 子选项 然后重启supervisor.
- 可能部分节点网络配置错误. 查看 worker进程在启动时崩溃,没有堆栈跟踪 的解决方案

## Topology在一段时间后停止处理tuple

现象:

- 处理工作正常工作一段时间,然后突然停止,并且spout tuple开始大量失败.

解决方案:

- 这是ZeroMQ 2.1.10的已知问题. 降级至ZeroMQ 2.1.7

## Storm UI中supervisor节点显示缺失

现象:

- Storm UI 查看部分 supervisor进程缺失
- Storm UI 在刷新时supervisors 列表会变化

解决方案:

- 确保supervisor本地目录是独立的（例如,不通过NFS共享本地目录）
- 尝试删除supervisor的本地目录并重新启动守护进程. supervisor启动时为自己创建一个唯一的ID,并将其存储在本地. 当该id被复制到其他节点时,Storm会感到困惑.

## "Multiple defaults.yaml found" 错误

现象:

- 当使用 "storm jar" 部署topology时发生上述错误

解决方案:

- 很有可能在您的topology jar中包含有Storm相关jar. 当打包 topology jar时, 不要包含 Storm jars ,Storm 会自动将相关的jar包加入classpath中.

## 运行storm jar 发生 "NoSuchMethodError"

现象:

- 当运行 storm jar发生奇怪的 "NoSuchMethodError"错误

解决方案:

- 您正在使用与构建topology不同的Storm版本来部署topology. 确保您使用的Storm客户端与您编译topology的版本相同

## Kryo 并发修改异常(ConcurrentModificationException)

现象:

- 运行时异常堆栈跟踪如下:

```
java.lang.RuntimeException: java.util.ConcurrentModificationException
  at org.apache.storm.utils.DisruptorQueue.consumeBatchToCursor(DisruptorQueue.java:84)
  at org.apache.storm.utils.DisruptorQueue.consumeBatchWhenAvailable(DisruptorQueue.java:55)
  at org.apache.storm.disruptor$consume_batch_when_available.invoke(disruptor.clj:56)
```

```
at org.apache.storm.disruptor$consume_loop_STAR$_fn__1597.invoke(disruptor.clj:67)
at org.apache.storm.util$async_loop$fn__465.invoke(util.clj:377)
at clojure.lang.AFn.run(AFn.java:24)
at java.lang.Thread.run(Thread.java:679)
Caused by: java.util.ConcurrentModificationException
at java.util.LinkedHashMap$LinkedHashIterator.nextEntry(LinkedHashMap.java:390)
at java.util.LinkedHashMap$EntryIterator.next(LinkedHashMap.java:409)
at java.util.LinkedHashMap$EntryIterator.next(LinkedHashMap.java:408)
at java.util.HashMap.writeObject(HashMap.java:1016)
at sun.reflect.GeneratedMethodAccessor17.invoke(Unknown Source)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
at java.lang.reflect.Method.invoke(Method.java:616)
at java.io.ObjectStreamClass.invokeWriteObject(ObjectStreamClass.java:959)
at java.io.ObjectOutputStream.writeSerialData(ObjectOutputStream.java:1480)
at java.io.ObjectOutputStream.writeOrdinaryObject(ObjectOutputStream.java:1416)
at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1174)
at java.io.ObjectOutputStream.writeObject(ObjectOutputStream.java:346)
at org.apache.storm.serialization.SerializableSerializer.write(SerializableSerializer.java:21)
at com.esotericsoftware.kryo.Kryo.writeClassAndObject(Kryo.java:554)
at com.esotericsoftware.kryo.serializers.CollectionSerializer.write(CollectionSerializer.java:7)
at com.esotericsoftware.kryo.serializers.CollectionSerializer.write(CollectionSerializer.java:1)
at com.esotericsoftware.kryo.Kryo.writeObject(Kryo.java:472)
at org.apache.storm.serialization.KryoValuesSerializer.serializeInto(KryoValuesSerializer.java:
```

III

## 解决方案:

- 这意味着您将一个可变对象作为 **output tuple** 发出. 发送到 **output collector** 的一切对象都必须是不可变的. 当对象被序列化并通过网络发送时您的 **bolt** 同时正在修改对象.

# 在生产集群上运行 Topology

在生产集群上运行 Topology 类似于在 [本地模式](#) 下运行.以下是步骤:

- 1) 定义 Topology (如果使用 Java 定义, 则使用 [TopologyBuilder](#) )
- 2) 使用 [StormSubmitter](#) 将 topology 提交到集群. [StormSubmitter](#) 以 topology 的名称, topology 的配置和 topology 本身作为输入.例如:

```
Config conf = new Config();
conf.setNumWorkers(20);
conf.setMaxSpoutPending(5000);
StormSubmitter.submitTopology("mytopology", conf, topology);
```

- 3) 创建一个包含你的代码和代码的所有依赖项的 jar (除了 Storm - Storm jar 将被添加到 worker 节点上的 classpath 中) .

如果您使用 [Maven](#), [Maven Assembly Plugin](#) 插件可以为您做包装.只需将其添加到您的 pom.xml 中即可:

```
<plugin>
  <artifactId>maven-assembly-plugin</artifactId>
  <configuration>
    <descriptorRefs>
      <descriptorRef>jar-with-dependencies</descriptorRef>
    </descriptorRefs>
    <archive>
      <manifest>
        <mainClass>com.path.to.main.Class</mainClass>
      </manifest>
    </archive>
  </configuration>
</plugin>
```

然后运行 mvn assembly:assembly 来获取适当打包的 jar. 确保您 [排除了](#) Storm jar, 因为群集已经在类路径上有 Storm.

- 4) 使用 [storm](#) 客户端将 topology 提交到集群, 指定您的 jar 的路径, 要运行的类名以及将使用的任何参数:

```
storm jar path/to/allmycode.jar org.me.MyTopology arg1 arg2 arg3
```

[storm jar](#) 将 jar 提交到集群并配置 [StormSubmitter](#) 该类与正确的集群进行通信.在这个例子中, 上传 jar 后 [storm jar](#) , [org.me.MyTopology](#) 使用参数 "arg1", "arg2", and "arg3" 调用 main 函数.

您可以找到如何配置 `storm` 客户端与 Storm 集群进行交流, 以 [设置开发环境](#).

## 常用配置

您可以根据 `topology` 设置各种配置. 您可以在 [这里](#) 找到您可以设置的所有配置的列表. 以 "TOPOLOGY" 为前缀的可以在 `topology` 特定的基础上被覆盖（其他的是集群配置, 不能被覆盖）. 以下是为 `topology` 设置的一些常见的:

1. **Config.TOPOLOGY\_WORKERS:** 设置用于执行 `topology` 的 `worker` 进程数. 例如, 如果将其设置为25, 则集群中将有25个 Java 进程执行所有任务. 如果 `topology` 中的所有组件都具有150个并行度, 则每个 `worker` 进程将在其中运行6个任务作为线程.
2. **Config.TOPOLOGY\_ACKER\_EXECUTORS:** 这将设置跟踪元组树 `executor` 的数量, 并检测出 `spout` 元组何时完全处理. `Ackers` 是 Storm 可靠性模型的组成部分, 您可以在 [保证消息处理](#) 中阅读更多信息. 通过不设置此变量或将其设置为 `null`, Storm 将 `executor` 的数量设置为等于为此 `topology` 配置的 `worker` 数. 如果这个变量设置为0, 那么 Storm 会立即从元器件脱落出来, 使其可靠性降低.
3. **Config.TOPOLOGY\_MAX\_SPOUT\_PENDING:** 这将一次设置单个 `spout` 任务中可以挂起的 `spout` 元组的最大数量（挂起意味着元组尚未被确认或失败）. 强烈建议您设置此配置以防止队列爆炸.
4. **Config.TOPOLOGY\_MESSAGE\_TIMEOUT\_SECS:** 这是一个 `spout` 元组在被认为失败之前必须完全完成的最长时间. 此值默认为30秒, 这对大多数拓扑结构都是足够的. 有关 Storm 的可靠性模型如何工作的更多信息, 请参阅 [保证消息处理](#).
5. **Config.TOPOLOGY\_SERIALIZATIONS:** 您可以使用此配置向 Storm 注册更多序列化程序, 以便您可以在元组内使用自定义类型.

## Killing 一个 topology

要 kill 一个 topology, 只需运行:

```
storm kill {stormname}
```

提供与 `storm kill` 提交 `topology` 时使用的名称相同的名称.

Storm不会立即杀死 topology. 相反, 它会停用所有的端口, 以使它们不再发出任何元组, 然后 Storm 会在销毁所有 workers 之前等待 Config.

`TOPOLOGY_MESSAGE_TIMEOUT_SECS` 秒. 这给了 `topology` 足够的时间来完成它被杀死时处理的任何元组.

## Updating 一个正在运行的 topology

要更新正在运行的 topology, 目前唯一的选项是终止当前 topology 并重新提交新的 topology. 一个计划的功能是实现一个 `storm swap` 交换正在运行的 topology 结构的命令, 以确保最短的停机时间, 并且两个 topology 不会同时处理元组.

## Monitoring topologies

监控 topology 的最佳位置是使用 Storm UI. Storm UI 提供有关每个运行 topology 的每个组件的吞吐量和延迟性能的任务和精细统计信息中发生的错误的信息.

您还可以查看群集机器上的 worker 日志.

# Maven

要开发 topology, 您将需要您的类路径上的 Storm jar. 您应该在项目的类路径中包含未打包的 jar, 或者使用 Maven 将 Storm 包含为开发依赖项. Storm 托管在 Maven Central. 要将 Storm 作为开发依赖项包含在项目中, 请将以下内容添加到 pom.xml 中:

```
<dependency>
  <groupId>org.apache.storm</groupId>
  <artifactId>storm-core</artifactId>
  <version></version>
  <scope>provided</scope>
</dependency>
```

这是 Storm 项目的 pom.xml [示例](#).

发展 Storm

详情请参考 [DEVELOPER.md](#).

# 安全地运行 Apache Storm

## 安全地运行 Apache Storm

尝试保护集群时, Apache Storm 提供了一系列配置选项. 默认情况下, 所有验证和授权都被禁用, 但可以根据需要打开.

### 防火墙/操作系统 级别安全

您仍然可以拥有安全的 `storm` 集群, 而无需开启正式的身份验证和授权. 但是这样做通常需要配置您的操作系统来限制可以完成的操作. 即使您计划使用 `Auth` 运行群集, 通常也是一个好主意.

如何设置这些预防措施的具体细节有很大的不同, 超出了本文档的范围.

通常情况下, 启用防火墙并限制传入网络连接仅限于源自群集本身和来自可信主机和服务的网络连接, 一个完整的端口 `storm` 使用列表如下.

如果您的群集正在处理的数据是敏感的, 则可能最好设置 `IPsec` 来加密群集中主机之间发送的所有流量.

## Ports

默认端口	Storm 配置	Client Hosts/进程	Server
2181	<code>storm.zookeeper.port</code>	Nimbus, Supervisors, and Worker 进程	Zookeeper
6627	<code>nimbus.thrift.port</code>	Storm clients, Supervisors, and UI	Nimbus
8080	<code>ui.port</code>	Client Web 浏览器	UI
8000	<code>logviewer.port</code>	Client Web 浏览器	Logviewer
3772	<code>drpc.port</code>	外部 DRPC Clients	DRPC
3773	<code>drpc.invocations.port</code>	Worker 进程	DRPC
3774	<code>drpc.http.port</code>	外部 HTTP DRPC Clients	DRPC
670{0,1,2,3}	<code>supervisor.slots.ports</code>	Worker 进程	Worker 进程

## UI / Logviewer

UI 和 logviewer 进程提供了一种方法, 不仅可以看到集群正在做什么, 还可以操纵运行的 topology. 通常, 这些进程不应该被暴露, 除了群集的用户.

通常使用某种形式的身份验证, 使用 java servlet 过滤器.

```
ui.filter: "filter.class"
ui.filter.params: "param1":"value1"
```

或通过限制 UI / Logviewer 端口仅接受来自本地主机的连接, 然后使用另一个可以验证/授权传入连接并代理与 storm 进程的连接的 Web 服务器 (如: Apache httpd) 将其连接起来. 为了使此工作, ui 进程必须将 logviewer.port 设置为其 storm.yaml 中的代理端口, 而日志查看器必须将其设置为要绑定到的实际端口.

servlet 过滤器是首选, 因为它允许单独的 topology 来详细说明谁是谁, 谁不被允许访问与它们相关联的页面.

Storm UI 可以配置为使用 hadoop-auth 中的 AuthenticationFilter.

```
ui.filter: "org.apache.hadoop.security.authentication.server.AuthenticationFilter"
ui.filter.params:
  "type": "kerberos"
  "kerberos.principal": "HTTP/nimbus.witzend.com"
  "kerberos.keytab": "/vagrant/keytabs/http.keytab"
  "kerberos.name.rules": "RULE:[2:$1@$0]([jt]t@.*EXAMPLE.COM)s/.*/$MAPRED_USER/ RULE:[2:$1@$0]([nd]
```

确保创建一个主体 'HTTP/{hostname}' (这里的 hostname 应该是运行 UI 守护进程的主机名)

一旦配置用户需要在访问 UI 之前执行 kinit. 例如:

```
curl -i --negotiate -u:anyUser -b ~/cookiejar.txt -c ~/cookiejar.txt http://storm-ui-hostname:8080/ai
```

1. Firefox: 转到 配置并搜索 network.negotiate-auth.trusted-uris 双击以添加值 "<http://storm-ui-hostname:8080>"
2. Google-chrome: 从命令行开始 google-chrome --auth-server-whitelist="\*storm-ui-hostname" --auth-negotiate-delegate-whitelist="\*storm-ui-hostname"
3. IE: 配置受信任的网站以包含 "storm-ui-hostname" 并允许该网站的协商

注意: 为了通过 logviewer 安全模式查看任何日志, 运行的所有主机 logviewer 也应该添加到上

述白名单中.对于大集群, 您可以列出主机的域 (例如: 设置 `network.negotiate-auth.trusted-uris` 为 `.yourdomain.com`) .

警告: 在 AD MIT Keberos 设置中, 密钥大小大于默认的 UI jetty 服务器请求头大小. 确保在 `storm.yaml` 中将 `ui.header.buffer.bytes` 设置为 65536 . 更多详细信息, 请参见 [STORM-633](#)

## UI / DRPC SSL

UI 和 DRPC 都允许用户配置 ssl.

### UI

对于 UI 用户需要在 `storm.yaml` 中设置以下配置. 在此步骤之前, 用户必须注意使用合适的密钥和证书生成密钥库.

1. `ui.https.port`
2. `ui.https.keystore.type` (示例: "jks")
3. `ui.https.keystore.path` (示例: "/etc/ssl/storm\_keystore.jks")
4. `ui.https.keystore.password` (密钥库密码)
5. `ui.https.key.password` (私钥密码)

可选配置

1. `ui.https.truststore.path` (示例: "/etc/ssl/storm\_truststore.jks")
2. `ui.https.truststore.password` (信任密码)
3. `ui.https.truststore.type` (示例: "jks")

如果用户想要设置双向认证

1. `ui.https.want.client.auth` (如果这设置为客户端认证身份验证的真实服务器请求, 但如果  
没有提供身份验证, 则保持连接)
2. `ui.https.need.client.auth` (如果设置为 `true` 服务器需要客户端提供身份验证)

### DRPC

与 UI 类似, 用户需要为 DRPC 配置以下内容

1. `drpc.https.port`
2. `drpc.https.keystore.type` (示例: "jks")

3. drpc.https.keystore.path (示例: "/etc/ssl/storm\_keystore.jks")
4. drpc.https.keystore.password (密钥库密码)
5. drpc.https.key.password (私钥密码)

可选配置

1. drpc.https.truststore.path (示例: "/etc/ssl/storm\_truststore.jks")
2. drpc.https.truststore.password (信任密码)
3. drpc.https.truststore.type (示例: "jks")

如果用户想要设置双向认证

1. drpc.https.want.client.auth (如果这设置为客户端证书认证的真实服务器请求, 但如果沒有提供认证, 则保持连接)
2. drpc.https.need.client.auth (如果设置为 true 服务器需要客户端提供身份验证)

## 认证 (**Kerberos**)

Storm 通过 thrift 和 SASL 提供可插拔的身份验证支持. 此示例仅适用于 Kerberos , 因为它是大多数大型数据项目的常见设置.

在每个节点上设置 KDC 并配置 kerberos 超出了本文档的范围, 并假设您已经完成了.

### 创建 Headless Principals and keytabs

每个 Zookeeper 服务器, Nimbus 和 DRPC 服务器将需要一个服务的 principal, 按照惯例, 它将包含将运行的主机的 FQDN. 请注意, zookeeper 用户必须是 zookeeper.supervisor 和 UI 还需要一个 principal 来运行, 但由于它们是外向连接, 所以他们不需要是服务的 principal. 以下是如何设置 kerberos principal 的示例, 但细节可能会因您的 KDC 和操作系统而异.

```
# Zookeeper (Will need one of these for each box in teh Zk ensamble)
sudo kadmin.local -q 'addprinc zookeeper/zk1.example.com@STORM.EXAMPLE.COM'
sudo kadmin.local -q "ktadd -k /tmp/zk.keytab zookeeper/zk1.example.com@STORM.EXAMPLE.COM"
# Nimbus and DRPC
sudo kadmin.local -q 'addprinc storm/storm.example.com@STORM.EXAMPLE.COM'
sudo kadmin.local -q "ktadd -k /tmp/storm.keytab storm/storm.example.com@STORM.EXAMPLE.COM"
# All UI logviewer and Supervisors
sudo kadmin.local -q 'addprinc storm@STORM.EXAMPLE.COM'
sudo kadmin.local -q "ktadd -k /tmp/storm.keytab storm@STORM.EXAMPLE.COM"
```

确保将keytab分发到相应的框, 并设置 FS 权限, 以便只有运行 ZK 或 storm 的 headless 用户才能访问它们.

## Storm Kerberos 配置

Storm 和 Zookeeper 都使用 jaas 配置文件来记录用户. 每个 jaas 文件可能有不同的界面被使用.

要在 storm 中启用 Kerberos 身份验证, 您需要设置以下 storm.yaml 配置:

```
storm.thrift.transport: "org.apache.storm.security.auth.kerberos.KerberosSaslTransportPlugin"  
java.security.auth.login.config: "/path/to/jaas.conf"
```

Nimbus 和 supervisor 进程也将连接到 ZooKeeper(ZK), 我们希望将其配置为使用 Kerberos 进行身份验证. 做这个附加:

```
-Djava.security.auth.login.config=/path/to/jaas.conf
```

对 nimbus , ui 和 supervisor 的 childdopts.给出了写入时, 默认的 childdopts 设置的一个例子:

```
nimbus.childdopts: "-Xmx1024m -Djava.security.auth.login.config=/path/to/jaas.conf"  
ui.childdopts: "-Xmx768m -Djava.security.auth.login.config=/path/to/jaas.conf"  
supervisor.childdopts: "-Xmx256m -Djava.security.auth.login.config=/path/to/jaas.conf"
```

对于 storm 节点, jaas.conf 文件应如下所示. StormServer 部分由 nimbus 和 DRPC 节点使用. 它不需要包括在主管节点上. StormClient 部分被所有想要与 nimbus 通讯的 storm client 使用, 包括 ui, logviewer 和 supervisor. 我们将在网关上使用这一部分, 但其结构将会有不同. Client 部分被想要与 zookeeper 通讯的进程使用, 并且只需要包含在 nimbus 和 supervisor 中. 服务器部分由 zookeeper 服务器使用. 在 jaas 中没有使用的部分不是问题.

```
StormServer {  
    com.sun.security.auth.module.Krb5LoginModule required  
    useKeyTab=true  
    keyTab="$keytab"  
    storeKey=true  
    useTicketCache=false  
    principal="$principal";  
};  
StormClient {  
    com.sun.security.auth.module.Krb5LoginModule required  
    useKeyTab=true  
    keyTab="$keytab"  
    storeKey=true  
    useTicketCache=false  
    serviceName="$nimbus_user"  
    principal="$principal";  
};  
Client {  
    com.sun.security.auth.module.Krb5LoginModule required  
    useKeyTab=true
```

```

keyTab="$keytab"
storeKey=true
useTicketCache=false
serviceName="zookeeper"
principal="$principal";
};

Server {
    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true
    keyTab="$keytab"
    storeKey=true
    useTicketCache=false
    principal="$principal";
};

}

```

以下是基于 keytab 生成的示例:

```

StormServer {
    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true
    keyTab="/keytabs/storm.keytab"
    storeKey=true
    useTicketCache=false
    principal="storm/storm.example.com@STORM.EXAMPLE.COM";
};

StormClient {
    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true
    keyTab="/keytabs/storm.keytab"
    storeKey=true
    useTicketCache=false
    serviceName="storm"
    principal="storm@STORM.EXAMPLE.COM";
};

Client {
    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true
    keyTab="/keytabs/storm.keytab"
    storeKey=true
    useTicketCache=false
    serviceName="zookeeper"
    principal="storm@STORM.EXAMPLE.COM";
};

Server {
    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true
    keyTab="/keytabs/zk.keytab"
    storeKey=true
    useTicketCache=false
    serviceName="zookeeper"
    principal="zookeeper/zk1.example.com@STORM.EXAMPLE.COM";
};

```

Nimbus 还会将 principal 转换成本地用户名, 以便其他服务可以使用此名称.配置 Kerberos 身份验证集

```
storm.principal.tolocal: "org.apache.storm.security.auth.KerberosPrincipalToLocal"
```

这只需要在 `nimbus` 上完成, 但不会对任何节点造成伤害. 我们还需要从 `ZooKeeper` 的角度通知 `topology` 谁是 `supervisor` 守护程序和 `nimbus` 守护程序正在运行.

```
storm.zookeeper.superACL: "sasl:$\{nimbus-user\}"
```

这里 `nimbus-user` 是 `nimbus` 用于使用 `ZooKeeper` 进行身份验证的 `Kerberos` 用户. 如果 `ZooKeeper` 正在剥离主机和领域, 那么这需要主机和领域也被剥离.

## ZooKeeper 集成

关于如何设置安全的 `ZK` 的完整细节超出了本文档的范围. 但是, 一般来说, 您要在每个服务器上启用 `SASL` 身份验证, 并可选择剥离主机和领域.

```
authProvider.1 = org.apache.zookeeper.server.auth.SASLAuthenticationProvider
kerberos.removeHostFromPrincipal = true
kerberos.removeRealmFromPrincipal = true
```

并且您希望在启动服务器时在命令行中包含 `jaas.conf`, 以便可以使用它来查找 `keytab`.

```
-Djava.security.auth.login.config=/jaas/zk_jaas.conf
```

## 网关

理想情况下, 最终用户只需要在与 `storm` 交互之前运行 `kinit`. 为了无缝地实现这一点, 我们需要在网关上使用默认的 `jaas.conf`.

```
StormClient {
    com.sun.security.auth.module.Krb5LoginModule required
    doNotPrompt=false
    useTicketCache=true
    serviceName="$nimbus_user";
};
```

如果最终用户具有无头键的用户, 则可以覆盖此 `keytab`.

## 授权设置

身份验证 完成了验证用户是谁的工作, 但是我们也需要 授权 来执行每个用户可以执行的任务.

`nimbus` 的首选授权插件是 `SimpleACLAutorizer`. 要使用 `SimpleACLAutorizer`, 请设置以下

内容:

```
nimbus.authorizer: "org.apache.storm.security.auth.authorizer.SimpleACLAuthorizer"
```

DRPC 有一个独立的授权器配置. 不要对 DRPC 使用 SimpleACLAuthorizer.

该 *SimpleACLAuthorizer* 插件需要知道谁主管用户, 它需要知道所有的管理员用户, 包括运行UI守护进程在用户的.

这些通过 *nimbus.supervisor.users* 和 *nimbus.admins* 分别设置. 每个可以是一个完整的 Kerberos principal 名称, 也可以是主机和领域的用户名.

日志服务器有自己的授权配置. 这些都是通过 *logs.users* 和 *logs.groups* 设置的. 应将这些设置为集群中所有节点的管理员用户或组.

提交 *topology* 时, 提交用户也可以在此列表中指定用户. 指定的用户和组（除了群集范围内的用户）将被授予对日志查看器中提交的 *topology* 的 *worker* 日志的访问权限.

## Supervisors 匿名用户 和 组设置

为了确保 multi-tenancy 用户的隔离, 需要在 supervisor 节点上运行 supervisors, 匿名用户和 唯一组 来执行. 要启用以下步骤.

1. 向所有 supervisor 主机添加 headlessuser.
2. 创建唯一的组, 并使其成为 supervisor 节点上 匿名用户的主组.
3. 对于这些 supervisor 节点, 设置以下 storm 属性.

## Multi-tenant 调度

为了更好地支持 multi-tenancy, 我们已经编写了一个新的调度程序. 启用此调度程序集.

```
storm.scheduler: "org.apache.storm.scheduler.multitenant.MultitenantScheduler"
```

请注意, 此调度程序的许多功能都依赖于 storm 身份验证. 没有他们, 调度程序将不知道用户是什么, 也不会正确地隔离 topology.

multi-tenant 调度程序的目标是提供一种将 topology 彼此隔离的方法, 但也限制了个人用户可以在群集中拥有的资源.

调度器当前有一个配置, 可以通过 =storm.yaml= 或通过一个名为 =multitenant-

`scheduler.yaml`= 的单独配置文件来设置, 该配置文件应该放在与 `=storm.yaml`= 相同的目录中. 最好使用 `=multitenant-scheduler.yaml`=, 因为它可以更新而不需要重新启动 `nimbus`.

当前只有一个配置 `=multitenant-scheduler.yaml`=, `=multitenant.scheduler.user.pools`= 是从用户名的映射到用户保证能够用于其 `topology` 结构的最大节点数.

例如:

```
multitenant.scheduler.user.pools:  
  "evans": 10  
  "derek": 10
```

以提交 `topology` 的用户身份运行 `worker` 进程

默认情况下, `storm` 运行作为运行 `supervisor` 的用户的 `worker`. 这不是安全的理想选择. 使 `storm` 作为启动它们的用户进行 `topology` 运行.

```
supervisor.run.worker.as.user: true
```

有几个与此相关的文件需要正确配置以使 `storm` 安全.

`worker-launcher` 可执行文件是一个特殊程序, 允许 `supervisor` 以不同的用户身份启动 `worker`. 为了这个工作, 它需要由 `root` 拥有, 但是该组被设置为只有 `supervisor` 匿名用户是其中的一部分的组. 它还需要拥有 6550 个权限. 还有一个 `worker-launcher.cfg` 文件, 通常位于 `/etc/` 下, 应该如下所示:

```
storm.worker-launcher.group=$(worker_launcher_group)  
min.user.id=$(min_user_id)
```

其中 `worker_launcher_group` 是 `supervisor` 其中一部分的同一组, 并且 `min.user.id` 设置为系统上的第一个真实用户标识. 此配置文件也需要由 `root` 拥有, 不具有世界或组写权限.

冒充一个用户

`storm client` 可以代表另一个用户提交请求. 例如, 如果 `userX` 提交 `oozie` 工作流程, 并且作为工作流执行的一部分; 如果用户 `oozie` 想要代表 `userX` 它提交 `topology`, 可以通过利用模拟功能来实现. 为了提交 `topology` 作为其他用户, 您可以使用 `StormSubmitter.submitTopologyAs` API. 或者, 您可以使用 `NimbusClient.getConfiguredClientAs` `nimbus client` 作为其他用户, 并使用此 `client` 执行任何 `nimbus` 操作 (即: `kill/rebalance/activate/deactivate`) .

默认情况下禁用模拟授权, 这意味着任何用户都可以执行模拟. 为了确保只有授权用户可以执行模拟, 您应该启动 `nimbus.nimbus.impersonation.authorizer` 设置

`org.apache.storm.security.auth.authorizer.ImpersonationAuthorizer`. 该 `ImpersonationAuthorizer` 用于 `nimbus.impersonation.acl` 为 ACL 对用户进行授权. 以下是用于支持模拟的 `nimbus` 配置示例:

```
nimbus.impersonation.authorizer: org.apache.storm.security.auth.authorizer.ImpersonationAuthorizer
nimbus.impersonation.acl:
    impersonating_user1:
        hosts:
            [comma separated list of hosts from which impersonating_user1 is allowed to impersonate
        groups:
            [comma separated list of groups whose users impersonating_user1 is allowed to impersonate
    impersonating_user2:
        hosts:
            [comma separated list of hosts from which impersonating_user2 is allowed to impersonate
        groups:
            [comma separated list of groups whose users impersonating_user2 is allowed to impersonate
```

为了支持 `oozie` 用例, 可以提供以下配置:

```
nimbus.impersonation.acl:
    oozie:
        hosts:
            [oozie-host1, oozie-host2, 127.0.0.1]
        groups:
            [some-group-that-userX-is-part-of]
```

## 自动凭证的推送和更新

个人 `topology` 能够向 `worker` 推送凭证（票证和令牌）, 以便他们可以访问安全服务. 将它暴露给所有的用户可能会对它们造成痛苦. 要在常见情况下隐藏这些插件, 可以使用插件来填充凭据, 将另一方解压缩到 `java` 主题中, 并且还允许 `Nimbus` 在需要时更新凭据. 这些由以下配置控制. `topology.auto-credentials` 是一个 `java` 插件的列表, 所有这些插件都必须实现 `IAutoCredentials` 接口, 该接口填充网关上的凭据, 并在 `worker` 端解包它们. 在 `kerberos` 安全集群上, 默认情况下应设置为指向 `org.apache.storm.security.auth.kerberos.AutoTGT`. `nimbus.credential.renewers.classes` 也应设置为此值, 以便 `nimbus` 可以代表用户定期更新 `TGT`.

`nimbus.credential.renewers.freq.secs` 控制更新者轮询多长时间查看是否需要更新, 但默认值应该是正常的.

此外, `Nimbus` 本身可以用于代表用户提交 `topology` 来获取凭据. 这可以使用

`nimbus.autocredential.plugins.classes` 进行配置, 这是完全限定类名的列表, 所有这些都必须实现 `INimbusCredentialPlugin`. Nimbus 将调用所有配置的实现的 `populateCredentials` 方法作为 `topology` 提交的一部分. 您应该使用此配置与 `topology.auto-credentials` 和 `nimbus.credential.renewers.classes`, 以便凭据可以在 `worker` 端填充, 并且 `nimbus` 可以自动更新它们. 目前有两个使用此配置的示例, `AutoHDFS` 和 `AutoHBase`, 它们自动填充 `topology` 提交程序的 `hdfs` 和 `hbase` 委托令牌, 以便他们不必在所有可能的工作主机上分发密钥表.

## 范围

默认情况下, `storm` 允许提交任何大小的 `topology`. 但 ZK 等人对 `topology` 结构实际上有多大的限制. 以下配置允许您限制 `topology` 的最大大小.

YAML 设置	描述
<code>nimbus.slots.perTopology</code>	<code>topology</code> 可以使用的 <code>slots/workers</code> 的最大数量.
<code>nimbus.executors.perTopology</code>	<code>topology</code> 可以使用的最大 <code>executors/threads</code> .

## 日志清理

`LogViewer` 守护进程现在也负责清理旧的日志文件, 以防止 `dead topologies`.

YAML 设置	描述
<code>logviewer.cleanup.age.mins</code>	<code>worker</code> 的日志必须在该日志被考虑进行清理之前（最后修改时间）多大. (生活 <code>worker</code> 的日志永远不会被 <code>logviewer</code> 清理: 他们的日志通过回拨滚动.)
<code>logviewer.cleanup.interval.secs</code>	日志记录器清理工作日志的时间间隔（秒）.

## 允许特定用户或组访问 `storm`

使用 `SimpleACLAuthorizer`, 任何具有有效 `Kerberos` 票证的用户都可以部署 `topology` 或进行其他操作, 例如 激活, 停用, 访问集群信息. 可以通过指定 `nimbus.users` 或 `nimbus.groups` 来限制此访问. 如果 `nimbus.users` 仅配置列表中的用户可以部署 `topology` 或访问集群. 类似地, `nimbus.groups` 限制对属于这些组的用户的 `storm` 集群访问.

要配置, 请在 `storm.yaml` 中指定以下配置:

```
nimbus.users:  
  - "testuser"
```

或者

```
nimbus.groups:  
  - "storm"
```

## DRPC

希望更多在这个很快

# CGroup Enforcement

## CGroups in Storm

Storm 使用 CGroup 来限制 worker 的资源使用, 以保证公平和 QOS.

请注意: **CGroups** 目前仅支持 **Linux** 平台 (内核版本 **2.6.24** 及更高版本)

## 设置

要使用 CGroups, 请确保正确安装 cgroups 并配置 cgroup. 有关设置和配置的更多信息, 请访问:

[https://access.redhat.com/documentation/en-US/Red\\_Hat\\_Enterprise\\_Linux/6/html/Resource\\_Management\\_Guide/ch-Using\\_Control\\_Groups.html](https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Resource_Management_Guide/ch-Using_Control_Groups.html)

一个示例/默认的 cgconfig.conf 文件 /conf 目录. 内容如下:

```
mount {  
    cpuset  = /cgroup/cpuset;  
    cpu   = /cgroup/storm_resources;  
    cpuacct = /cgroup/cpuacct;  
    memory  = /cgroup/storm_resources;  
    devices  = /cgroup/devices;  
    freezer  = /cgroup/freezer;  
    net_cls  = /cgroup/net_cls;  
    blkio    = /cgroup/blkio;  
}  
  
group storm {  
    perm {  
        task {  
            uid = 500;  
            gid = 500;  
        }  
        admin {  
            uid = 500;  
            gid = 500;  
        }  
    }  
    cpu {  
    }  
}
```

有关 cgconfig.conf 文件的格式和配置的更详细说明, 请访问:

[https://access.redhat.com/documentation/en-US/Red\\_Hat\\_Enterprise\\_Linux/6/html/Resource\\_Management\\_Guide/ch-Using\\_Control\\_Groups.html](https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Resource_Management_Guide/ch-Using_Control_Groups.html)

## 与 Storm 中的 CGroup 相关的设置

设置	功能
storm.cgroup.enable	此配置用于设置是否使用 cgroup. 设置 "true" 以后 "false" 不使用 cgroups. 当该配置设置为 false 时, 单元测试. 默认设置为 "false"
storm.cgroup.hierarchy.dir	到风暴将使用的 cgroup 层次结构的路径. 默认设置 "/cgroup/storm_resources"
storm.cgroup.resources	将由 CGroups 监管的子系统列表. 默认设置为 cpu 和内存
storm.supervisor.cgroup.rootdir	supervisor 使用的根 cgroup.cgroup 的路径将是 <storm.cgroup.hierarchy.dir>/<storm.supervisor.cgroup.rootdir> 为 "storm"
storm.cgroup.cgexec.cmd	用于在 cgroup 中启动工作的 cgexec 命令的绝对 "/bin/cgexec"
storm.worker.cgroup.memory.mb.limit	每个 worker 的内存限制为 MB. 这可以基于每个 supervisor 节点上所有 worker 进程的 CPU 使用量, 请设置 config supervisor.cpu.capacity. 其中每个增量代表核心的 1%, 因此如果用户设置 supervisor.cpu.capacity: 200, 则用户指示使用 2 个内核.
storm.worker.cgroup.cpu.limit	每个 worker 的 CPU 份额. 这可以基于每个 supervisor 节点上所有 worker 进程的 CPU 使用量, 请设置 config supervisor.cpu.capacity. 其中每个增量代表核心的 1%, 因此如果用户设置 supervisor.cpu.capacity: 200, 则用户指示使用 2 个内核.

由于通过 config supervisor.cpu.capacity 限制 CPU 使用率仅限制进程的 CPU 占用比例, 以限制 supervisor 节点上所有 worker 进程的 CPU 使用量, 请设置 config supervisor.cpu.capacity. 其中每个增量代表核心的 1%, 因此如果用户设置 supervisor.cpu.capacity: 200, 则用户指示使用 2 个内核.

CGroup 可以与资源意识调度程序一起使用. 然后, CGroup 将强制资源意识调度程序分配的 worker 的资源使用情况. 要将资源感知计划程序使用 cgroup, 只需启用 cgroups, 并确保不要设置 storm.worker.cgroup.memory.mb.limit 和 storm.worker.cgroup.cpu.limit 配置.

# Pacemaker

## 介绍

Pacemaker 是一个旨在处理 **worker** 心跳的 **storm** 守护进程. 随着 **storm** 的扩大, ZooKeeper 由于 **worker** 进行心跳的大量写入而开始成为瓶颈. 当 ZooKeeper 尝试维护一致性时, 会产生大量写入磁盘和跨网络的流量.

因为心跳是短暂的, 它们不需要被持久化到磁盘或跨节点同步; 会在内存◆的存储中来做. 这是 Pacemaker 的作用. Pacemaker 作为一个简单的 **key/value** 存储, 具有类似 ZooKeeper 的目录样式键和字节数组值.

相应的 Pacemaker client 是 `ClusterState` 接口的插件

`org.apache.storm.pacemaker.pacemaker_state_factory`. 心跳调用由漏斗 `ClusterState` 由生产 `pacemaker_state_factory` 到 Pacemaker 服务进程, 而另一 `set/get` 操作被转发到 ZooKeeper.

## 配置

- `pacemaker.host` : (已弃用) Pacemaker 守护程序正在运行的主机
- `pacemaker.servers` : Pacemaker 守护程序正在运行的主机 - 这取代了 `pacemaker.host`
- `pacemaker.port` : Pacemaker 监听端口
- `pacemaker.max.threads` : Pacemaker 守护进程将用于处理请求的最大线程数.
- `pacemaker.childopts` : 任何需要转到 Pacemaker 的 JVM 参数. (由 **storm-deploy** 项目使用)
- `pacemaker.auth.method` : 使用的身份验证方法 (以下更多信息)

## 示例

要使 Pacemaker 启动并运行, 请在所有节点上的集群配置中设置以下选项:

```
storm.cluster.state.store: "org.apache.storm.pacemaker.pacemaker_state_factory"
```

Pacemaker 服务器还需要在所有节点上设置:

```
pacemaker.servers:  
  - somehost.mycompany.com  
  - someotherhost.mycompany.com
```

`pacemaker.host` 配置仍适用于单个 pacemaker, 尽管它已被弃用.

```
pacemaker.host: single_pacemaker.mycompany.com
```

然后启动所有的守护进程(包括 Pacemaker):

```
$ storm pacemaker
```

Storm 集群现在应该通过 Pacemaker 来推动所有 worker 的心跳.

## 安全

目前支持摘要（基于密码）和 **Kerberos** 安全性. 安全性目前只在于读取而不是写入. 写入可以由任何人执行, 而读取只能由授权和认证的用户执行. 这是未来发展的一个领域, 因为它让群集开放给 DoS 攻击, 但它阻止任何敏感信息到达未经授权的眼睛, 这是主要目标.

## Digest

要配置摘要身份验证, 请 `pacemaker.auth.method: DIGEST` 在集群配置中设置托管 Nimbus 和 Pacemaker 的节点. 节点也必须 `java.security.auth.login.config` 设置为指向包含以下结构的 JAAS 配置文件:

```
PacemakerDigest {  
    username="some username"  
    password="some password";  
};
```

配置了这些设置的任何节点将能够从 Pacemaker 中读取. Worker 节点不需要设置这些配置, 并且可以保留 `pacemaker.auth.method: NONE` 设置, 因为它们不需要从 Pacemaker 守护进程读取.

## Kerberos

要配置Kerberos身份验证, 请 `pacemaker.auth.method: KERBEROS` 在主机 Nimbus 和 Pacemaker 的节点上的集群配置中进行设置. 节点也必须 `java.security.auth.login.config` 设置为指向 JAAS 配置.

Nimbus 上的 JAAS 配置必须看起来像这样:

```
PacemakerClient {  
    com.sun.security.auth.module.Krb5LoginModule required  
    useKeyTab=true  
    keyTab="/etc/keytabs/nimbus.keytab"  
    storeKey=true
```

```
useTicketCache=false  
serviceName="pacemaker"  
principal="nimbus@MY.COMPANY.COM";  
};
```

Pacemaker 上的 JAAS 配置必须如下所示：

```
PacemakerServer {  
    com.sun.security.auth.module.Krb5LoginModule required  
    useKeyTab=true  
    keyTab="/etc/keytabs/pacemaker.keytab"  
    storeKey=true  
    useTicketCache=false  
    principal="pacemaker@MY.COMPANY.COM";  
};
```

- Nimbus 主机上 PacemakerClient 部分中的客户端用户主体必须与 Storm 集群上的 nimbus.daemon.user 配置值相匹配.
- 在, 客户端的 serviceName 值必须与 Pacemaker 主机上 PacemakerServer 部分的服务器的用户主体相匹配.

容错

Pacemaker 作为单个守护进程运行, 使其成为潜在的单点故障.

如果 Pacemaker 由 Nimbus 无法通过崩溃或网络分区, worker 将继续运行, Nimbus 将重复尝试重新连接. Nimbus 的功能将受到干扰, 但 topology 本身将继续运行. 如果 Nimbus 和 Pacemaker 位于分区同一侧的集群分区, 分区另一侧的 worker 将无法心跳, Nimbus 将重新安排其他任务. 这可能是我们想要发生的事情.

## ZooKeeper 比较

与 ZooKeeper 相比, Pacemaker 使用更少的 CPU, 更少的内存, 当然也没有磁盘用于相同的负载, 这是由于缺乏维护节点之间的一致性的开销. 在千兆网络上, 有6000个节点的理论限制. 然而, 实际限制可能在2000-3000节点之间. 这些限制还没有被测试. 在拥有 topology 结构的270个管理员集群中, Pacemaker 的资源利用率是一个核心的70%, 在具有4个 Intel(R) Xeon(R) CPU E5530 @ 2.40GHz 和 24GiB RAM 的机器上的近 1GiB 的 RAM.

Pacemaker 在支持 HA. 多个 Pacemaker 实例可以在 Storm 群集中一次使用, 以实现大规模的可扩展性. 只需将 Pacemaker 主机的名称包含在 pacemaker.servers 配置中, worker 和 Nimbus 将开始与他们进行通信. 他们也是容错的. 只要至少有一个 Pacemaker 运行, 系统就

会继续工作 - 只要它可以处理负载.

# 资源感知调度器 (Resource Aware Scheduler)

## 介绍

本文档的目的是为 Storm 分布式实时计算系统提供资源感知调度程序的描述。 本文档将为您提供 Storm 中资源感知调度程序的高级描述。 以下是 Hadoop Summit 2016 演示文稿中概述的一些好处是在 Storm 上使用资源感知调度器：

<http://www.slideshare.net/HadoopSummit/resource-aware-scheduling-in-apache-storm>

## Table of Contents

1. 使用资源感知调度器
2. API 概述
  1. 设置内存要求
  2. 设置 CPU 要求
  3. 设置每个 worker (JVM) 进程的堆大小
  4. 在节点上设置可用资源
  5. 其他配置
3. Topology 优先级和每个用户资源
  1. 设置
  2. 指定 Topology 优先级
  3. 指定 Scheduling 策略
  4. 指定 Topology 优先策略
  5. 指定 Eviction 策略
4. 分析资源使用情况
5. 对原始 DefaultResourceAwareStrategy 的增强

## 使用资源感知调度器

用户可以通过在 `conf/storm.yaml` 中设置以下内容来切换到使用资源感知调度器

```
storm.scheduler: "org.apache.storm.scheduler.resource.ResourceAwareScheduler"
```

## API 概述

要使用 Trident， 请参阅 [Trident RAS API](#)

对于 **Storm Topology**，用户现在可以指定运行组件的单个实例所需的 **Topology** 组件（即：**Spout** 或 **Bolt**）的资源量。 用户可以通过使用以下 **API** 调用来指定 **Topology** 组件的资源需求。

## 设置内存要求

**API** 设置组件内存要求：

```
public T setMemoryLoad(Number onHeap, Number offHeap)
```

参数：

- **Number onHeap** - 此组件的实例将以兆字节消耗的堆内存量
- **Number offHeap** - 此组件的一个实例将以兆字节消耗的堆内存量

用户还必须选择只要指定堆内存要求，如果组件没有关闭堆内存需要。

```
public T setMemoryLoad(Number onHeap)
```

参数：

- **Number onHeap** - 此组件的一个实例将占用的堆内存量

如果没有为 **offHeap** 提供值，将使用 0.0。 如果没有为 **onHeap** 提供任何值，或者从未为 **◆◆** 调用的组件 **API**，则将使用默认值。

使用示例：

```
SpoutDeclarer s1 = builder.setSpout("word", new TestWordSpout(), 10);
s1.setMemoryLoad(1024.0, 512.0);
builder.setBolt("exclaim1", new ExclamationBolt(), 3)
    .shuffleGrouping("word").setMemoryLoad(512.0);
```

该 **topology** 结构请求的整个内存为 16.5 GB。这是从10个 **spout**，堆内存为 1GB，每个堆内存为0.5 GB，每个堆内存为3个 **bolt** 0.5 GB。

## 设置 **CPU** 要求

设置组件 **CPU** 要求的 **API**：

```
public T setCPUload(Double amount)
```

参数:

- Number amount – 该组件实例将使用的CPU数量

目前，一个组件需要或在节点上可用的 CPU 资源量由一个 point 系统表示。CPU 使用是一个难以定义的概念。根据手头的任务，不同的 CPU 架构执行不同。它们非常复杂，在单个精确的便携式数字中表达所有这些都是不可能的。相反，我们采取了一种配置方法的惯例，主要关注 CPU 使用率的粗略水平，同时仍然提供了指定更细粒度的可能性。

按照惯例，CPU 内核通常会得到100分。如果您觉得您的处理器或多或少功能强大，您可以相应地进行调整。CPU 绑定的重任务将获得100分，因为它们可以消耗整个内核。中等任务应该得到50，轻型任务25和小任务10。在某些情况下，您有一个任务可以产生其他线程来帮助处理。这些任务可能需要超过100点才能表达他们正在使用的 CPU 数量。如果遵循这些约定，单个线程任务的常见情况，报告的 Capacity \* 100应该是任务需要的 CPU 点数。

使用示例:

```
SpoutDeclarer s1 = builder.setSpout("word", new TestWordSpout(), 10);
s1.setCPUload(15.0);
builder.setBolt("exclaim1", new ExclamationBolt(), 3)
    .shuffleGrouping("word").setCPUload(10.0);
builder.setBolt("exclaim2", new HeavyBolt(), 1)
    .shuffleGrouping("exclaim1").setCPUload(450.0);
```

设置每个 **worker (JVM)** 进程的堆大小

```
public void setTopologyWorkerMaxHeapSize(Number size)
```

参数:

- Number size – worker 进程将以兆字节来分配内存范围

在每个 topology 基础上分配给单个 worker 程序的内存资源量，用户可以通过使用上述 API 来限制资源感知调度器。该 API 已经到位，以便用户可以将 executor 传播给多个 worker。然而，将 executor 传播给多个 worker 可能会增加通信延迟，因为 executor 将无法使用 Disruptor Queue 进行进程内通信。

使用示例:

```
Config conf = new Config();
```

```
conf.setTopologyWorkerMaxHeapSize(512.0);
```

在节点上设置可用资源

storm 管理员可以通过修改位于该节点的 `storm home` 目录中的 `conf/storm.yaml` 文件来指定节点资源可用性。

storm 管理员可以指定一个节点有多少可用内存（兆字节），将以下内容添加到 `storm.yaml` 中

```
supervisor.memory.capacity.mb: [amount<Double>]
```

storm 管理员还可以指定节点有多少可用 CPU 资源，将以下内容添加到 `storm.yaml` 中

```
supervisor.cpu.capacity: [amount<Double>]
```

注意：用户可以为可用 CPU 指定的数量使用如前所述的点系统来表示。

使用示例：

```
supervisor.memory.capacity.mb: 20480.0  
supervisor.cpu.capacity: 100.0
```

其他配置

用户可以在 `conf/storm.yaml` 中为资源意识调度程序设置一些默认配置：

```
//default value if on heap memory requirement is not specified for a component  
topology.component.resources.onheap.memory.mb: 128.0  
  
//default value if off heap memory requirement is not specified for a component  
topology.component.resources.offheap.memory.mb: 0.0  
  
//default value if CPU requirement is not specified for a component  
topology.component.cpu.pcure.percent: 10.0  
  
//default value for the max heap size for a worker  
topology.worker.max.heap.size.mb: 768.0
```

## Topology 优先级和每个用户资源

资源感知调度器或 RAS 还具有 multitenant 功能，因为许多 Storm 用户通常共享 Storm 集群。 资源感知调度器可以在每个用户的基础上分配资源。 每个用户可以保证一定数量的资源来运行他或她的 topology，并且资源感知调度器将尽可能满足这些保证。当 Storm 群集具有额外的免费资源时，资源感知调度器将能够以公平的方式为用户分配额外的资源。

**topology** 的重要性也可能有所不同。 **topology** 可用于实际生产或仅用于实验，因此资源感知调度器将在确定调度 **topology** 的顺序或何时驱逐 **topology** 时考虑 **topology** 的重要性。

## 设置

可以指定用户的资源保证 *conf/user-resource-pools.yaml*。以下列格式指定用户的资源保证：

```
resource.aware.scheduler.user.pools:  
[UserId]  
  cpu: [Amount of Guarantee CPU Resources]  
  memory: [Amount of Guarantee Memory Resources]
```

*user-resource-pools.yaml* 可以是什么样的示例：

```
resource.aware.scheduler.user.pools:  
  jerry:  
    cpu: 1000  
    memory: 8192.0  
  derek:  
    cpu: 10000.0  
    memory: 32768  
  bobby:  
    cpu: 5000.0  
    memory: 16384.0
```

请注意，指定数量的保证 CPU 和内存可以是整数或双倍

## 指定 **Topology** 优先级

**topology** 优先级的范围可以从0-29开始。**topology** 优先级将被划分为可能包含一系列优先级的几个优先级。例如，我们可以创建一个优先级映射：

```
PRODUCTION => 0 - 9  
STAGING => 10 - 19  
DEV => 20 - 29
```

因此，每个优先级包含10个子优先级。用户可以使用以下 API 设置 **topology** 的优先级

```
conf.setTopologyPriority(int priority)
```

参数：\* **priority** - 表示 **topology** 优先级的整数

请注意，0-29范围不是硬限制。因此，用户可以设置高于29的优先级数。然而，优先级数越高的属性越低，重要性仍然保持不变

## 指定 **Scheduling** 策略

用户可以在每个 **topology** 基础上指定要使用的调度策略。 用户可以实现 **IStrategy** 界面，并定义新的策略来安排特定的 **topology**。 这个可插拔接口是因为我们实现不同的 **topology** 可能具有不同的调度需求而创建的。 用户可以使用 API 在 **topology** 定义中设置 **topology** 策略：

```
public void setTopologyStrategy(Class<? extends IStrategy> clazz)
```

参数：

- **clazz** - 实现 **IStrategy** 接口的策略类

使用示例：

```
conf.setTopologyStrategy(org.apache.storm.scheduler.resource.strategies.scheduling.DefaultResourceA
```

提供默认调度。**DefaultResourceAwareStrategy** 是基于 Storm 中的资源感知调度原始文件中的调度算法实现的：

Peng, Boyang, Mohammad Hosseini, Zhihao Hong, Reza Farivar, 和 Roy Campbell。 "R-storm: storm 中的资源感知调度"。 在第16届年度中间件会议论文集，第149-161页。 ACM, 2015。

<http://dl.acm.org/citation.cfm?id=2814808>

**Please Note:** 必须根据本文所述的原始调度策略进行增强。请参阅"原始 **DefaultResourceAwareStrategy** 的增强功能"一节"

## 指定 **Topology** 优先策略

调度顺序是可插拔接口，用户可以在其中定义 **topology** 优先级的策略。为了使用户能够定义自己的优先级策略，他或她需要实现 **ISchedulingPriorityStrategy** 界面。 用户可以通过将 **Config.RESOURCE\_AWARE\_SCHEDULER\_PRIORITY\_STRATEGY** 设置为指向实现策略的类来设置调度优先级策略。 例如：

```
resource.aware.scheduler.priority.strategy: "org.apache.storm.scheduler.resource.strategies.priorit
```

将提供默认策略。以下说明默认调度优先级策略的工作原理。

## DefaultSchedulingPriorityStrategy

调度顺序应基于用户当前资源分配与其保证分配之间的距离。我们应优先考虑远离资源保障的用户。这个问题的难点在于，用户可能有多个资源保证，另一个用户可以拥有另一套资源保证，那么我们怎么能以公平的方式来比较呢？我们用平均百分比的资源担保作为比较方法。

例如：

用户	资源保证	资源分配
A	<10 cpu="" 50gb="">	<2 40="" cpu="" gb="">
B	< 20 CPU, 25GB>	<15 10="" cpu="" gb="">

用户 A 的平均百分比满足资源保证：

$$(2/10+40/50)/2 = 0.5$$

用户 B 的资源保证的平均百分比满足：

$$(15/20+10/25)/2 = 0.575$$

因此，在该示例中，用户 A 具有比用户 B 满足的资源保证的平均百分比如较小。因此，用户 A 应优先分配更多资源，即调度用户 A 提交的 topology。

在进行调度时，RAS 按用户资源保证和平均百分比满足资源保证的平均百分比，按用户的平均百分比满足资源保证的平均百分比排序，根据用户的顺序排序。当用户的资源保证完全满足时，用户满足资源保证的平均百分比大于等于1。

## 指定 Eviction 策略

当集群中没有足够的可用资源来安排新的 topology 结构时，使用 eviction 策略。如果集群已满，我们需要一种 eviction topology 的机制，以便满足用户资源保证，并且可以在用户之间公平分享其他资源。驱逐 topology 的策略也是可插拔的界面，用户可以在其中实现自己的 topology eviction 策略。为了使用户实现自己的 eviction 策略，他或她需要实现 IEvictionStrategy 接口并将

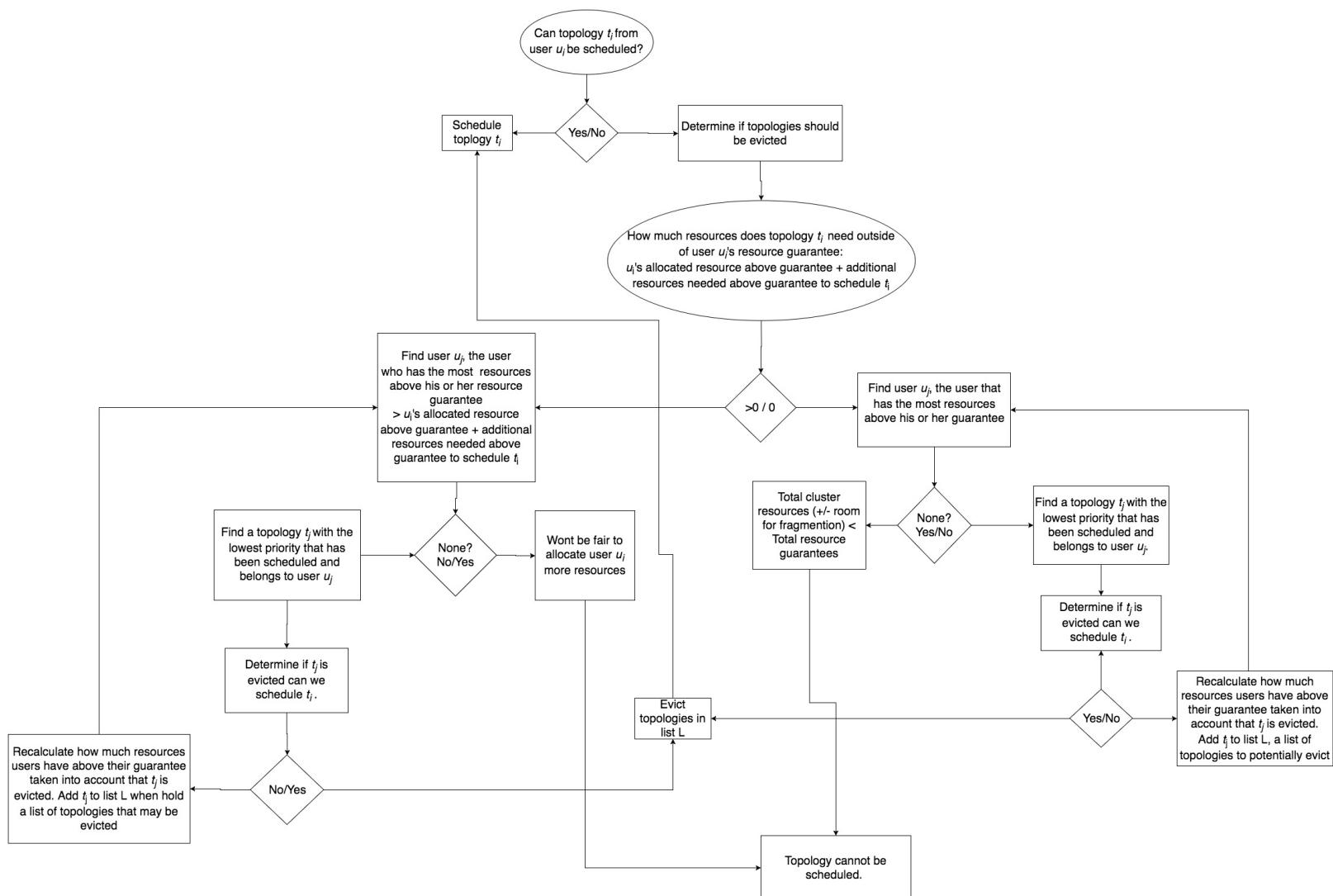
`Config.RESOURCE_AWARE_SCHEDULER_EVICTION_STRATEGY` 设置为指向已实施的策略类。例如：

提供了默认的 **eviction** 策略。以下说明默认 **topology eviction** 策略的工作原理

## DefaultEvictionStrategy

为了确定是否应该发生 **topology** 迁移，我们应该考虑到我们正在尝试调度 **topology** 的优先级，以及是否满足 **topology** 所有者的资源保证。

我们不应该从没有满足他或她的资源保证的用户中排除 **topology**。以下流程图应描述 **eviction** 过程的逻辑。



## 分析资源使用情况

了解 **topology** 的资源使用情况：

要了解 **topology** 结构实际使用的 memory/CPU 数量，您可以将以下内容添加到 **topology** 启动代码中。

```
//Log all storm metrics
conf.registerMetricsConsumer(backtype.storm.metric.LoggingMetricsConsumer.class);

//Add in per worker CPU measurement
Map<String, String> workerMetrics = new HashMap<String, String>();
workerMetrics.put("CPU", "org.apache.storm.metrics.sigar.CPUMetric");
conf.put(Config.TOPOLOGY_WORKER_METRICS, workerMetrics);
```

CPU 指标将需要您添加

```
<dependency>
  <groupId>org.apache.storm</groupId>
  <artifactId>storm-metrics</artifactId>
  <version>1.0.0</version>
</dependency>
```

作为 topology 依赖（1.0.0或更高版本）。

然后，您可以在 UI 上转到 topology，打开系统 metrics，并找到 LoggingMetricsConsumer 正在写入的日志。它会在日志中输出结果。

```
1454526100 node1.nodes.com:6707 -1:_system CPU {user-ms=74480, sys-ms=10780}
1454526100 node1.nodes.com:6707 -1:_system memory/nonHeap {unusedBytes=2077536, virtualFreeBytes=1970784}
1454526100 node1.nodes.com:6707 -1:_system memory/heap {unusedBytes=573861408, virtualFreeBytes=6152000}
```

-1: \_ 系统的度量通常是整个 worker 的 metrics 标准。 在上面的示例中，worker 正在 node1.nodes.com:6707 上运行。 这些 metrics 是每60秒收集一次。对于 CPU，您可以看到，在60秒钟内，此 worker 使用 $74480 + 10780 = 85260$  ms 的 CPU 时间。这相当于  $85260/60000$  或约1.5内核。

内存使用情况类似，但是查看 usedBytes。 offHeap 是 64621728 或大约 62MB，onHeap 是 83857888 或大约 80MB，但你应该知道你已经在每个 worker 中设置了你的堆。 你如何划分每个 bolt/spout？ 这有点困难，可能需要一些尝试和错误从你的结束。

## \* 对原始 **DefaultResourceAwareStrategy** 的增强 \*

如上文所述的默认资源感知调度策略有两个主要的调度阶段：

1. 任务选择 - 计算拓扑中的顺序 task/executor 应该被调度
2. 节点选择 - 给定一个 task/executor，找到一个节点来安排 task/executor。

对两个调度阶段进行了改进

## 任务选择增强

不是使用 **topology** 图的宽度优先遍历来创建组件及其 **executor** 的排序，而是使用一种新的启发式方法，可以通过组件的进出边缘数量（潜在连接）对组件进行排序。这被发现是一种更有效的方式来协调彼此通信的 **executor**，并减少网络延迟。

## 节点选择增强

节点选择首先选择哪个机架（服务器机架），然后选择该机架上的哪个节点。选择机架和节点的战略要点是找到具有“最多”资源的机架，并在该机架中使用“最多”免费资源查找节点。我们为此策略制定的假设是，拥有最多资源的节点或机架将具有最高的概率，允许我们调度在节点或机架上共同定位最多的 **executor**，以减少网络通信延迟。

机架和节点将从最佳选择排列到最差选择。在找到执行者时，策略将在放弃之前迭代所有机架和节点，从最坏到最坏。机架和节点将按以下事项进行排序：

1. 已经在机架或节点上安排了多少个 **executor** -- 这样做是为了使 **executor** 更紧密地安排已经安排并运行的 **executor**。如果 **topology** 部分崩溃，**topology** 的 **executor** 的一部分需要重新安排，我们希望将这些 **executor** 尽可能接近（网络）重新安排到健康和运行的 **executor**。
2. 辅助资源可用性或机架或节点上的“有效”资源量 -- 请参阅下属资源可用性部分
3. 所有资源可用性的平均值 -- 这仅仅是可用的平均百分比（分别在机架或集群上的可用资源分配的节点或机架上的可用资源）。只有当两个对象（机架或节点）的“有效资源”相同时，才会使用这种情况。然后，我们将所有资源百分比的平均值作为排序指标。例如：

```
Avail Resources:  
node 1: CPU = 50 Memory = 1024 Slots = 20  
node 2: CPU = 50 Memory = 8192 Slots = 40  
node 3: CPU = 1000 Memory = 0 Slots = 0
```

```
Effective resources for nodes:  
node 1 = 50 / (50+50+1000) = 0.045 (CPU bound)  
node 2 = 50 / (50+50+1000) = 0.045 (CPU bound)  
node 3 = 0 (memory and slots are 0)
```

节点1 和 节点2 具有相同的有效资源，但是明确地说，节点2具有比节点1更多的资源（**memory** 和 **slots**），并且我们将首先选择节点2，因为我们将能够安排更多的 **executor**。这是阶段2平均的

因此，排序遵循以下进展。基于1) 进行比较，如果相等，则基于2) 进行比较，如果基于3) 相等比较，并且如果相等，则通过基于比较节点或机架的 id 来通过任意分配排序来断开连接。

## 下属资源可用性

最初，RAS 的 `getBestClustering` 算法通过在机架中的所有节点上找到具有可用内存的最大可用总数 + 可用的最大可用机架，找到基于哪个机架具有 "最可用" 资源的 "最佳" 机架。这种方法不是非常准确的，因为内存和 `cpu` 的使用不同，而且值不是正常的。这种方法也没有效果，因为它不考虑可用的插槽的数量，并且由于资源之一（内存，CPU 或 slots）的耗尽，无法识别不可调度的机架。以前的方法也不考虑 `worker` 的失败。当 `topology` 的 `executor` 未被分配并需要重新安排时，

找到 "最佳" 机架或节点的新策略/算法，我配置从属资源可用性排序（受主导资源公平性的启发），通过下属（不占优势）资源可用性对机架和节点进行排序。

例如给出4个具有以下资源可用性的机架

```
//generate some that has a lot of memory but little of cpu
rack-3 Avail [ CPU 100.0 MEM 200000.0 Slots 40 ] Total [ CPU 100.0 MEM 200000.0 Slots 40 ]
//generate some supervisors that are depleted of one resource
rack-2 Avail [ CPU 0.0 MEM 80000.0 Slots 40 ] Total [ CPU 0.0 MEM 80000.0 Slots 40 ]
//generate some that has a lot of cpu but little of memory
rack-4 Avail [ CPU 6100.0 MEM 10000.0 Slots 40 ] Total [ CPU 6100.0 MEM 10000.0 Slots 40 ]
//generate another rack of supervisors with less resources than rack-0
rack-1 Avail [ CPU 2000.0 MEM 40000.0 Slots 40 ] Total [ CPU 2000.0 MEM 40000.0 Slots 40 ]
//best rack to choose
rack-0 Avail [ CPU 4000.0 MEM 80000.0 Slots 40 ] Total [ CPU 4000.0 MEM 80000.0 Slots 40 ]
Cluster Overall Avail [ CPU 12200.0 MEM 410000.0 Slots 200 ] Total [ CPU 12200.0 MEM 410000.0 Slots 200 ]
```

很明显，机架0是最平衡的最佳集群，可以安排最多的执行器，而机架2是机架2耗尽 `cpu` 资源的最差机架，因此即使有其他的可用资源

我们首先计算每个资源的所有机架的资源可用性百分比：

```
(resource available on rack) / (resource available in cluster)
```

我们做这个计算来归一化值，否则资源值将不可比较。

所以我们的例子：

```
rack-3 Avail [ CPU 0.819672131147541% MEM 48.78048780487805% Slots 20.0% ] effective resources: 0.0
```

```
rack-2 Avail [ 0.0% MEM 19.51219512195122% Slots 20.0% ] effective resources: 0.0
rack-4 Avail [ CPU 50.0% MEM 2.4390243902439024% Slots 20.0% ] effective resources: 0.0243902439024
rack-1 Avail [ CPU 16.39344262295082% MEM 9.75609756097561% Slots 20.0% ] effective resources: 0.09
rack-0 Avail [ CPU 32.78688524590164% MEM 19.51219512195122% Slots 20.0% ] effective resources: 0.1
```

机架的有效资源也是下属资源，计算方法如下：

```
MIN(resource availability percentage of {CPU, Memory, # of free Slots}).
```

然后我们用有效的资源订购机架。

因此我们的例子：

```
Sorted rack: [rack-0, rack-1, rack-4, rack-3, rack-2]
```

该 **metric** 用于对节点和机架进行排序。在分类机架时，我们考虑机架上和整个集群中可用的资源（包含所有机架）。在分类节点时，我们考虑节点上可用的资源和机架中可用的资源（机架中所有节点可用的所有资源的总和）

原始 Jira 为此增强: [STORM-1766](#)

### 计划改进

本节提供了一些关于性能优势的实验结果，其中包括在原始调度策略之上的增强功能。实验基于运行模拟：

<https://github.com/jerrypeng/storm-scheduler-test-framework>

模拟中使用随机 **topology** 和集群，以及由雅虎所有 **storm** 集群中运行的所有真实 **topology** 结构组成的综合数据集。

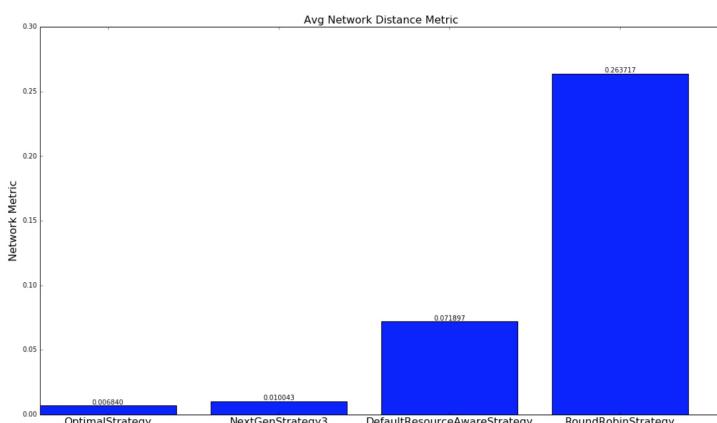
下图提供了各种策略调度 **topology** 结构以最小化网络延迟的比较。通过每个调度策略为 **topology** 的每个调度计算一个网络 **metric**。网络 **metric** 是根据 **topology** 中的每个 **executor** 对驻留在同一个工作程序（**JVM**进程）中的另一个 **executor**，在不同的 **worker** 但是相同的主机，不同的主机，不同的机架上进行的连接进行计算的。我们所做的假设如下：

1. Intra-worker 之间的沟通是最快的
2. Intra-worker 之间的沟通很快
3. Inter-node 间通信速度较慢
4. Inter-rack 间通信是最慢的

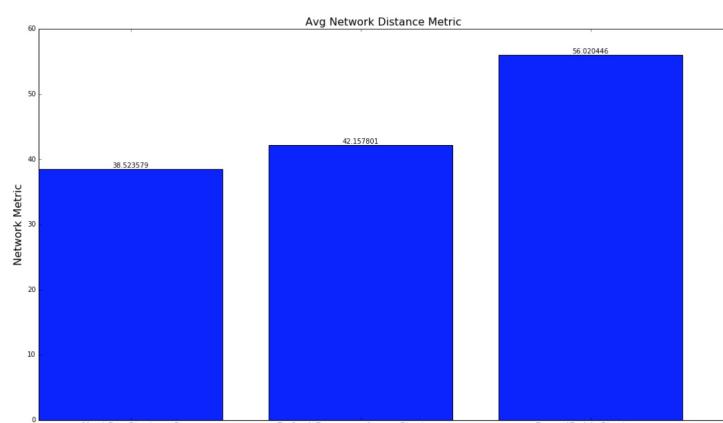
对于此网络 metric，数量越大，topology 结构对于此调度将具有更多的潜在网络延迟。进行两种实验。使用随机生成的 topology 进行一组实验，并随机生成簇。另一组实验使用包含基于 topology 大小的 yahoo 和 semi-randomly 生成的集群中的所有运行 topology 的数据集执行。两组实验都运行数百万次迭代，直到结果收敛。

对于涉及随机生成的 topology 结构的实验，实现了一种最优策略，如果存在解决方案，则会极大地找到最优解。本实验中使用的 topology 和簇相对较小，以便最优策略遍历解空间，以在合理的时间内找到最优解。由于 topology 很大，并且运行时间不合理，所以这种策略并不适用于雅虎 topology，因为解决方案空间是  $W^N$ （在工作中无关紧要），其中 W 是工作人员的数量，N 是执行者的数量。NextGenStrategy 代表具有这些增强功能的调度策略。DefaultResourceAwareStrategy 表示原始调度策略。RoundRobinStrategy 代表了一种 naive 策略，它简单地以循环方式安排执行者，同时遵守资源约束。下图显示了网络度量的平均值。CDF 图也进一步呈现。

Random Topologies



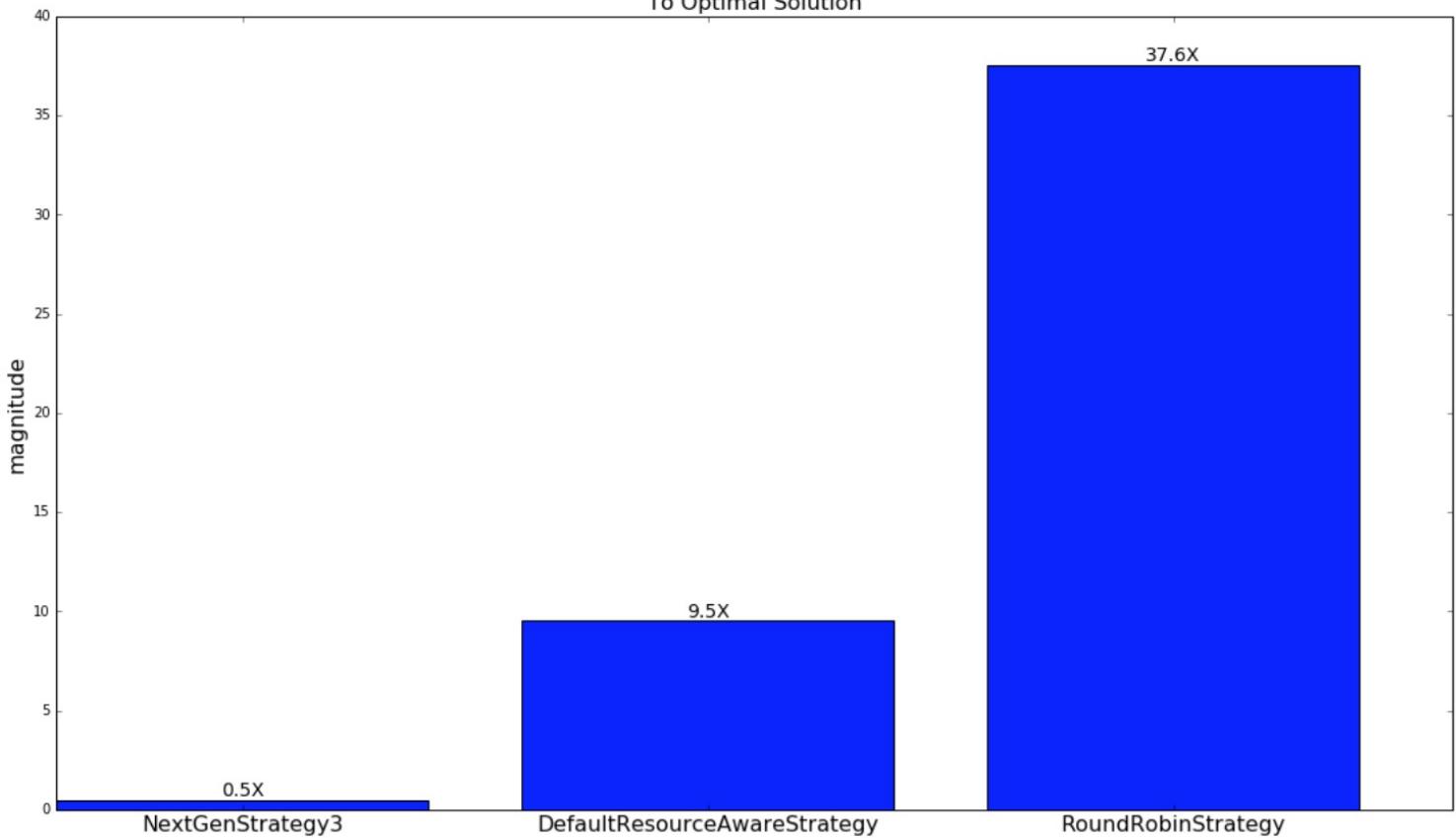
Yahoo topologies



下一个图表显示了从各自的调度策略的调度到最优策略的调度的接近程度。如前所述，这仅适用于随机生成的 topology 和集群。

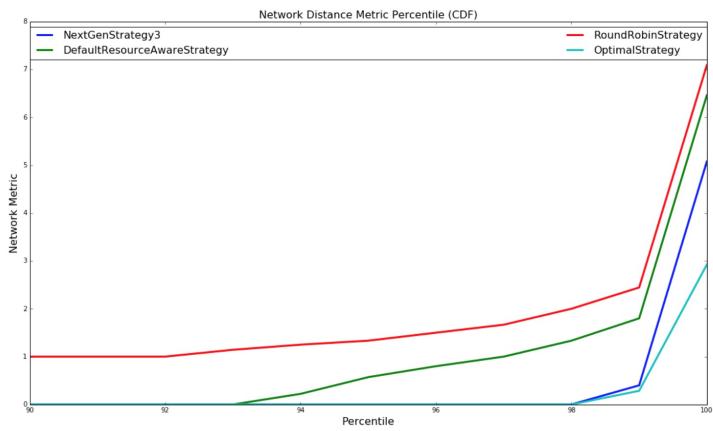
Random Topologies

### To Optimal Solution

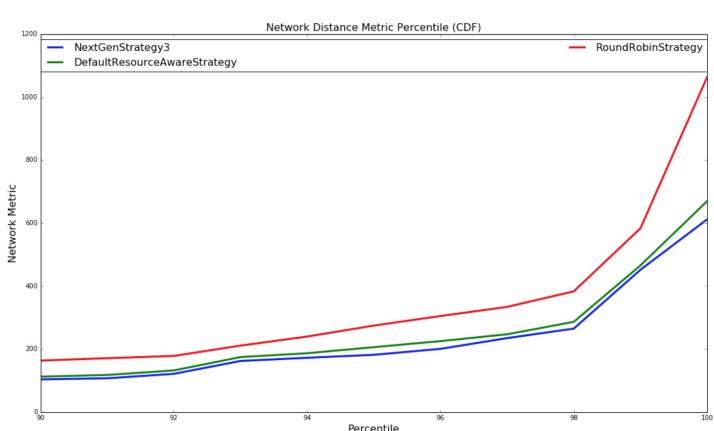


下图是网络 metric 的 CDF:

**Random Topologies**



**Yahoo topologies**

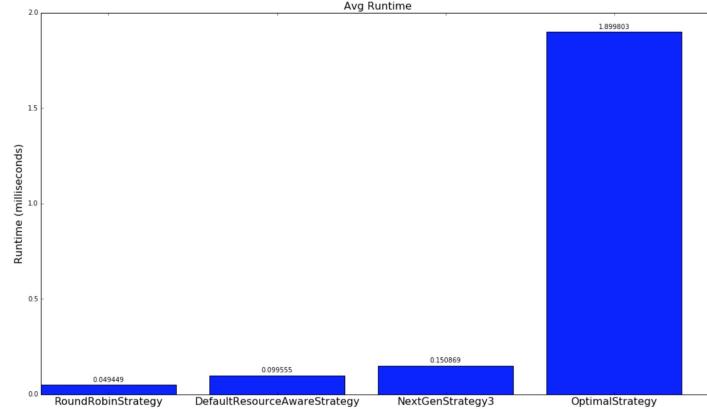


以下是策略运行多长时间的比较:

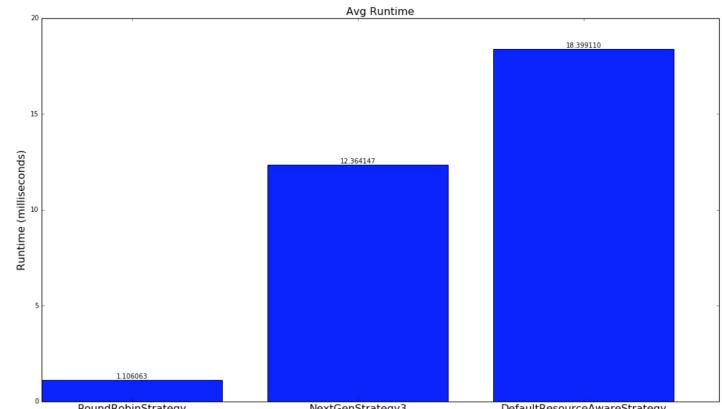
**Random Topologies**

**Yahoo topologies**

Avg Runtime



Avg Runtime



# 用于分析 **Storm** 的各种内部行为的 **Metrics**

随着这些 Metrics 的增加, Storm 用户可以收集, 查看和分析各种内部操作的性能. 分析的动作包括 Storm 守护程序中的 rpc 调用和 http 任务. 例如, 在 Storm Nimbus 守护进程中, 下面是在 `Nimbus$iface` 中定义的 thrift 调用简介:

- submitTopology
- submitTopologyWithOpts
- killTopology
- killTopologyWithOpts
- activate
- deactivate
- rebalance
- setLogConfig
- getLogConfig

各种 HTTP GET 和 POST 请求也被 make 用于分析, 例如 Storm UI 守护程序的 GET 和 POST 请求 (`ui/core.clj`) 要实现这些 Metrics, 使用以下软件包:

- `io.dropwizard.metrics`
- `metrics-clojure`

## 怎么运行它

通过使用包 `io.dropwizard.metrics` 和 `metrics-clojure` (Metrics Java API 的 clojure 包装器), 我们可以通过声明 `(defmeter num-some-func-calls)` 来将功能标记为配置文件, 然后添加 `(mark! num-some-func-calls)` 调用函数的位置. 例如:

```
(defmeter num-some-func-calls)
(defn some-func [args]
  (mark! num-some-func-calls)
  (body))
```

什么是 `mark!` 的本质! API 调用是增加一个计数器, 表示某个操作发生了多少次. 对于即时测量, 用户可以使用量规. 例如:

```
(defgauge nimbus:num-supervisors
  (fn [] (.size (.supervisors (:storm-cluster-state nimbus) nil))))
```

上面的例子将得到集群中的主管数量.这个度量不像以前讨论过的那样累积.

还需要激活度量报告服务器来收集指标.您可以通过调用以下函数来执行此操作:

```
(defn start-metrics-reporters []
  (jmx/start (jmx/reporter {})))
```

## 如何收集 Metrics

Metrics 可以通过 JMX 或 HTTP 报告.用户可以使用 JConsole 或 VisualVM 连接到 jvm 进程并查看统计信息.

要在 GUI 中查看 Metrics, 请使用 VisualVM 或 JConsole. 使用 VisualVm 进行 Metrics 的屏幕截图:



有关如何收集 Metrics 的详细信息, 请参考:

<https://dropwizard.github.io/metrics/3.1.0/getting-started/>

如果要使用 JMX 并通过 JConsole 或 VisualVM 查看 Metrics, 请记住使用正确的 JMX 配置启动要配置文件的 JVM 进程. 例如在 Storm 中, 您将添加以下 conf/storm.yaml

```
nimbus.childopts: "-Xmx1024m -Dcom.sun.management.jmxremote.local.only=false -Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremote.ssl=false"
ui.childopts: "-Xmx768m -Dcom.sun.management.jmxremote.port=3334 -Dcom.sun.management.jmxremote.local.only=false -Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremote.ssl=false"
logviewer.childopts: "-Xmx128m -Dcom.sun.management.jmxremote.port=3335 -Dcom.sun.management.jmxremote.local.only=false -Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremote.ssl=false"
drpc.childopts: "-Xmx768m -Dcom.sun.management.jmxremote.port=3336 -Dcom.sun.management.jmxremote.local.only=false -Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremote.ssl=false"
supervisor.childopts: "-Xmx256m -Dcom.sun.management.jmxremote.port=3337 -Dcom.sun.management.jmxremote.local.only=false -Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremote.ssl=false"
```

请注意:

由于我们遮蔽了我们使用的所有软件包, 所以用于收集 Metrics 的附加插件目前可能无法正常工作. 目前通过 JMX 收集 Metrics 是受支持的.

有关 io.dropwizard.metrics 和 metrics-clojure 软件包的更多信息, 请参考原始文档:

- <https://dropwizard.github.io/metrics/3.1.0/>
- <http://metrics-clojure.readthedocs.org/en/latest/>

# Windows 用户指南

本页介绍如何在 Windows 上为 Apache Storm 设置环境。

## 符号链接

从 1.0.0 开始，Apache Storm 符号链接 将日志目录和资源目录聚合到 `worker` 目录中。不幸的是，创建符号链接 在 Windows 上需要非默认权限，所以用户应该手动配置它，以确保 Storm 进程可以在运行时创建符号链接。根据 Windows 版本（即非专业版），由于安装了该工具，因此无法通过安全策略设置符号链接特权。

当创建符号链接不可能时，Supervisor 进程将在尝试启动 `worker` 时立即停止，因为权限异常被认为是致命的错误。

下面的页面 (MS technet) 指导如何将该策略配置到 Storm 运行的帐户。

- [如何配置安全策略设置](#)
- [创建符号链接](#)

一个棘手的一点是，`administrator` 组已经有了这个特权，但只有进程被激活才能作为 `administrator` 帐户运行。所以，如果您的帐户属于 `administrator` 组（和你不想改变它），你可能要打开 `command prompt` 与 `run as administrator` 和控制台内执行的过程。如果您不想直接执行 Storm 进程（而不是在命令提示符下），请执行 `runas /user:administrator` 以管理员帐户运行的进程。

从 Windows 10 创作者更新开始，可以激活支持创建符号链接的开发人员模式，而不以管理员身份运行 [Windows 10 中的符号链接！](#)

另外，您可以通过配置设置来禁用符号链接的使用`storm.disable.symlinks`，以`true` 在雨云和所有的主管节点。这也将禁用需要符号链接的功能。目前，这只是下载依赖的 Blob，但可能会在将来发生变化。一些拓扑结构可能依赖于作为便利创建的工作者的当前工作目录中的资源的符号链接，因此它不是100%向后兼容的更改。

或者，您可以通过在 Nimbus 和所有 Supervisor 节点上将 config `storm.disable.symlinks` 配置为 `true` 来禁用符号链接的使用。这也将禁用需要符号链接的功能。目前，这只是下载依赖的 Blob，但可能在将来发生变化。一些 topology 结构可能依赖于作为便利创建的工作者的当前工作目录中的资源的符号链接，因此它不是100%向后兼容的更改。

# Storm 中级

# 序列化

序列化 本文阐述了 Storm 0.6.0 以上版本的序列化机制。在低于 0.6.0 版本的 Storm 中使用了另一种序列化系统，详细信息可以参考 [Serialization \(prior to 0.6.0\)](#) 一文

Storm 中的 tuple 可以包含任何类型的对象。由于 Storm 是一个分布式系统，所以在不同的任务之间传递消息时 Storm 必须知道怎样序列化、反序列化消息对象。

Storm 使用 Kryo 对对象进行序列化。Kryo 是一个灵活、快速的序列化库。

Storm 默认支持基础类型、string、byte arrays、ArrayList、HashMap、HashSet 以及 Clojure 的集合类型的序列化。如果你需要在 tuple 中使用其他的对象类型，你就需要注册一个自定义的序列化器。

## Dynamic typing(动态类型)

在 tuple 中没有对各个字段 (field) 的直接类型声明。你需要将对象放入对应的字段中，然后 Storm 可以动态地实现对象的序列化。在学习序列化接口之前，我们先来了解一下为什么 Storm 的 tuple 是动态类型化的。

为 tuple fields 增加静态类型会大幅增加 Storm 的 API 的复杂度。比如 Hadoop 就将它的 key 和 value 都静态化了，这就要求用户自己添加大量的注解。使用 Hadoop 的 API 非常繁琐，而相应的“类型安全”不值得。相对的，动态类型就非常易于使用。

进一步说，也不可能有什么合理的方法将 Storm 的 tuple 的类型静态化。假如一个 Bolt 订阅了多个 stream，从这些 stream 传入的 tuple 很可能都带有不同的类型。在 Bolt 的 execute 方法接收到一个 tuple 的时候，这个 tuple 可能来自任何一个 stream，也可能包含各种组合类型。也许你可以使用某种反射机制来为 bolt 订阅的每个 tuple stream 声明一个方法类处理 tuple，但是 Storm 可以提供一种更简单、更直接的动态类型机制来解决这个问题。

最后，Storm 使用动态类型定义的另一个原因就是为了用简洁直观的方式使用 Clojure、JRuby 这样的动态类型语言。

## Custom serialization(自定义序列化)

前面已经提到，Storm 使用 Kryo 来处理序列化。如果要实现自定义的序列化生成器，你需要用 Kryo 注册一个新的序列化生成器。强烈建议读者先仔细阅读 [Kryo's home page](#) 来理解

它是怎样处理自定义的序列化的。

可以通过**topology**的 `topology.kryo.register` 属性来添加自定义序列化生成器。该属性接收一个注册器列表，每个注册项都可以使用以下两种注册格式中的一种格式： 1. 只有一个待注册的类的名称。在这种情况下，Storm 会使用 Kryo 的 `FieldsSerializer` 来序列化该类。这也许并不一定是该类的最优化方式 —— 可以查看 Kryo 的文档来了解更多细节内容。 2. 一个包含待注册的类的名称和实现了 `com.esotericsoftware.kryo.Serializer` 接口的类组成的集合。我们来看个例子。

```
topology.kryo.register:
- com.mycompany.CustomType1
- com.mycompany.CustomType2: com.mycompany.serializer.CustomType2Serializer
- com.mycompany.CustomType3
```

`com.mycompany.CustomType1` 和 `com.mycompany.CustomType3` 会使用 `FieldsSerializer`，而 `com.mycompany.CustomType2` 则会使用 `com.mycompany.serializer.CustomType2Serializer` 来实现序列化。

Storm provides helpers for registering serializers in a topology config. The `Config` class has a method called `registerSerialization` that takes in a registration to add to the config. 在 `topology` 的配置中，Storm 提供了用于注册序列化生成器的帮助类。`Config` 类有一个 `registerSerialization` 方法可以将序列化生成器注册到配置中。

`Config` 中有一个更高级的配置项做 `Config.TOPOLOGY_SKIP_MISSING_KRYO_REGISTRATIONS`。如果你将该项设置为 `true`，Storm 会忽略掉所有已注册但是在 `topology` 的类路径上没有相应的代码的序列化器。否则，Storm 会在无法查找到序列化器的时候抛出错误。如果你在集群中运行有多个 `topology` 并且每个 `topology` 都有不同的序列化器，但是你又想要在 `storm.yaml` 中声明好所有的序列化器，在这种情况下这个配置项会有很大的帮助。

## Java serialization(java 序列化)

如果 Storm 发现了一个没有注册序列化器的类型，它会使用 Java 序列化器来代替，如果这个对象无法被 Java 序列化器序列化，Storm 就会抛出异常。

注意，Java 自身的序列化机制非常耗费资源，而且不管在 CPU 的性能上还是在序列化对象的大小上都没有优势。强烈建议读者在生产环境中运行 `topology` 的时候注册一个自定义的序列化器。保留 Java 的序列化机制主要为了便于设计新 `topology` 的原型。

你可以通过将 `Config.TOPOLOGY_FALL_BACK_ON_JAVA_SERIALIZATION` 配置为 `false` 的方式来将序列化器回退到 Java 的序列化机制。

## Component-specific serialization registrations(特定组件的序列化注册)

Storm 0.7.0 支持对特定组件的配置（详情请参阅[Storm配置一文](#)）。当然，如果某个组件定义了一个序列化器，这个序列化器也需要能够支持其他的 bolt —— 否则，后续的 bolt 将会无法接收来自该组件的消息！

在提交topology 的时候，topology 会选择一组序列化器用于在所有的组件间传递消息。这是通过将特定组件的序列化器注册信息与普通的序列化器信息融合在一起实现的。如果两个组件为同一个类定义了两个序列化器，Storm 会从中任意选择一个。

如果在两个组件的序列化器注册信息冲突的时候需要强制使用一个序列化器，可以在 topology 级的配置中定义你想要的序列化器。对于序列化器的注册信息，拓扑中配置的值是优先于具体组件的配置的。

# 常见 Topology 模式

这一页列出了Storm topologies（拓扑）中的各种常见模式。

1. Batching
2. BasicBolt
3. In-memory caching + fields grouping combo (内存缓存+字段分组组合)
4. Streaming top N
5. TimeCacheMap for efficiently keeping a cache of things that have been recently updated(最近刚刚更新的 TimeCacheMap， 可以有效的存储缓存数据)
6. CoordinatedBolt and KeyedFairBolt for Distributed RPC(分布式RPC)

## Batching

通常情况下为了效率或者其他原因，你想成批处理一组元组而不是单独处理一组元组。例如，您希望对数据库进行批处理更新，或者进行某种类型的流聚合。

如果您希望在数据处理中具有可靠性，那么正确的方法是在实例等待 **bolt** 进行处理时保留实例变量中的元组。完成批处理操作后，再确认所持有的所有元组。

如果**bolt** 发出元组，那么您可能会想使用多锚来确保可靠性。这要看具体的应用程序而定。有关可靠性如何工作的详细信息，请参见[保证消息处理](#)。

## BasicBolt

许多**bolts** 遵循类似的阅读输入元组模式，基于输入元组发射零个或多个元组，然后在执行方法结束时会立即确认输入元组。与此模式匹配的**Bolts** 类似功能和过滤器。这是一个通用模式，**storm**会为你暴露一个称为[IBasicBolt](#) 接口，自动化模式。更多信息见[保证消息处理](#)。

## In-memory caching + fields grouping combo(在内存中缓存+字段分组组合)

在Storm bolts 中使用内存缓存很常见。当把它与**fields grouping** 相结合时，缓存会变得特别强大。例如，假设你有一个**bolt**，用于把短网址（如[bit.ly](#), [t.co](#), 等）扩展为长的网址。你可以通过保存短URL的LRU，来缓存长URL的扩展 避免做同样的HTTP请求 从而提高性能。假设组件“URLS”发出短URLS，组件“expand”将短URL扩展为长URL并在内部保留缓存。考虑下面两段代码之间的区别：

```
builder.setBolt("expand", new ExpandUrl(), parallelism)
    .shuffleGrouping(1);
```

```
builder.setBolt("expand", new ExpandUrl(), parallelism)
    .fieldsGrouping("urls", new Fields("url"));
```

第二种方法将有更有效的缓存，因为相同的URL将始终指向相同的任务。这避免了任务中缓存的重复，使得短URL更可能命中缓存。

## Streaming top N

一个常见的连续计算Storm 是通过“streaming top N”的某种排序来实现。假设有一个bolt，它会发射这种形式的元组["value", "count"]，并且您希望有一个基于顶部N元组的bolt来计数。要做到这一点，最简单的方法是有一个在流上执行全局组的bolt，并在内存中保存一个top N items列表。

显然，这种方法不能在较大的流中应用，因为整个流必须完成一个任务。一个更好的计算方法是在流的分区上并行地执行上面的多个N's，然后合并上面的N's个来得到全局的顶部N:

```
builder.setBolt("rank", new RankObjects(), parallelism)
    .fieldsGrouping("objects", new Fields("value"));
builder.setBolt("merge", new MergeObjects())
    .globalGrouping("rank");
```

这种模式之所以有效，是因为第一个bolt所做的字段分组，使您在语义上需要正确的区分。你可以在[storm-starter这里](#)中看到这个案例。

然而如果你想处理已知的数据倾斜问题，可以使用partialkeygrouping代替fieldsgrouping。他将分配两个downstream bolts 来分布式负载以替代个使用单独的一个。

```
builder.setBolt("count", new CountObjects(), parallelism)
    .partialKeyGrouping("objects", new Fields("value"));
builder.setBolt("rank", new AggregateCountsAndRank(), parallelism)
    .fieldsGrouping("count", new Fields("key"));
builder.setBolt("merge", new MergeRanksObjects())
    .globalGrouping("rank");
```

topology 需要一个额外的处理层来聚合来自上游螺栓的部分计数，但现在只处理聚合值，所以bolt 不受数据倾斜引起的影响。您可以在[storm-starter这里](#)中看到这种模式的示例。

**TimeCacheMap for efficiently keeping a cache of things that have been recently**

**updated** (最近刚刚更新的 **TimeCacheMap**，可以有效的存储缓存数据)

有时你想在最近活动的项目中保留一个缓存，并且让那些已经有一段时间没用的条目自动过期。**timecachemap**就是比较试用这个场景的数据结构，他提供了一种方式，可以在当你需要让条目过期时添加回调函数。

## **CoordinatedBolt and KeyedFairBolt for Distributed RPC** (分布式RPC coordinatedbolt和keyedfairbolt)

当在storm顶部构建分布式RPC应用程序时，通常需要两种常见模式。这些都是由在Storm codebase 中的“standard library”的**CoordinatedBolt**和**KeyedFairBolt** 封装的代码。

当你的bolt接收到任何给定请求的所有元组时，**CoordinatedBolt**封装了包含你的逻辑以及算法的bolt。它大量使用**direct streams** 来做到这一点。

**CoordinatedBolt**也封装了包含逻辑的bolt，并确保您的topology 同时处理多个DRPC调用，而不是一次一次连续的执行。

有关详细信息，请参阅[分布式 RPC](#)。

# Clojure DSL

Storm配有Clojure DSL，用于定义spouts(喷口)，bolts(螺栓)和topologies(拓扑)。Clojure DSL可以访问Java API暴露的所有内容，因此如果您是Clojure用户，您可以直接编写Storm拓扑，根本不需要使用Java。Clojure DSL的源码在[org.apache.storm.clojure](#)命名空间中定义。

本页概述了Clojure DSL的所有功能，包括：

1. Defining topologies(定义拓扑)
2. `defbolt`
3. `defspout`
4. Running topologies in local mode or on a cluster(在本地模式或集群上运行拓扑)
5. Testing topologies(测试拓扑)

## Defining topologies(定义拓扑)

请使用`topology`函来定义`topology`(拓扑)。`topology`有两个参数：“`spout specs(规格)`”的映射和“`bolt specs(规格)`”的映射。每个`spouts specs(规格)`和`bolt specs(规格)`通过指定输入和并行度 来将组件的代码连接到`topology`中。

我们来看一下[storm-starter](#) :项目中的`topology`定义示例：

```
(topology
 {"1" (spout-spec sentence-spout)
 "2" (spout-spec (sentence-spout-parameterized
                   ["the cat jumped over the door"
                    "greetings from a faraway land"])
                   :p 2)}
 {"3" (bolt-spec {"1" :shuffle "2" :shuffle}
                  split-sentence
                  :p 5)
 "4" (bolt-spec {"3" ["word"]}
                  word-count
                  :p 6)})
```

`spout`和`bolt specs(规格)`的映射是从组件ID到相应规格的映射。组件ID必须在映射上是唯一的。就像在Java中定义`topologies(拓扑)`一样，在声明`topologies(拓扑)`中的`bolts` 输入时使用的组件ID。

## spout-spec (spout-规格)

`spout-spec`作为`spout`实现（实现 [IRichSpout](#)的对像）的可选关键字参数的参数。当前存在的唯一选项是：`:p`选项，它指定了`spout`的并行性。如果您省略`:p`，则`spout`将作为单个任务执行。

## bolt-spec (bolt-规格)

`bolt-spec`作为`bolt`实现（实现[IRichBolt](#)的对像）的可选关键字参数的参数

输入声明是从`stream ids`到`stream groupings`的映射。`stream id` 可以有以下两种形式中的一种：

1. `[==component id== ==stream id==]`: Subscribes to a specific stream on a component(订阅组件上的特定流)
2. `==component id==`: Subscribes to the default stream on a component(订阅组件上的默认流)

`stream grouping`可以是以下之一：

1. `:shuffle`: 用`shuffle grouping`进行订阅
2. Vector of field names, like "`id`" "`namefields grouping`订阅指定的字段
3. `:global`: 使用`global grouping`进行订阅
4. `:all`: 使用`all grouping`进行订阅
5. `:direct`: 使用`direct grouping`进行订阅

See [Concepts](#) for more info on stream groupings. Here's an example input declaration showcasing the various ways to declare inputs: 有关`stream groupings`的更多信息，请参阅概念。下面是一个输入声明的示例，展示各种声明输入的方法：

```
{["2" "1"] :shuffle  
"3" ["field1" "field2"]  
["4" "2"] :global}
```

此输入声明共计三个流。它通过随机分组来订阅组件“2”上的流“1”，在字段“field1”和“field2”上以`fields grouping`的方式订阅组件“3”上的默认流，使用全局分组在组件“4”上订阅流“2”。

像`spout-spec`一样，`bolt-spec`唯一当前支持的关键字参数是：`p`，它指定了`bolt`的并行性。

## shell-bolt-spec (shell-bolt-规格)

`shell-bolt-spec` is used for defining bolts that are implemented in a non-JVM language. It takes as arguments the input declaration, the command line program to run, the name of the file implementing the bolt, an output specification, and then the same keyword arguments that `bolt-spec` accepts. `shell-bolt-spec` 用于定义以非JVM语言实现的bolts。它作为输入声明参数，在命令行程序中运行，用文件的名称实现bolt，输出规范以及接受的相同关键字参数作为参数的 `bolt-spec`。

这有一个`shell-bolt-spec`的例子：

```
(shell-bolt-spec {"1" :shuffle "2" ["id"]}
                 "python"
                 "mybolt.py"
                 ["outfield1" "outfield2"]
                 :p 25)
```

输出声明的语法在下面的`defbolt`部分中有更详细的描述。有关Storm的工作原理的详细信息，请参阅[使用Storm的非JVM语言](#)。

## defbolt

`defbolt` 用于在Clojure中定义bolts。这里对bolts有一个限制，那就是他必须是可序列化的，这就是为什么你不能仅仅具体化`IRichBolt`来实现一个bolts（closures不可序列化）。`defbolt`在这个限制的基础上为定义bolts提供了一种更好的语法，而不仅仅是实现一个Java接口的。

在最充分的表现形势下，`defbolt`支持参数化bolts，并在bolts执行期间保持关闭状态。它还提供了用于定义不需要额外功能的bolts的快捷方式。`defbolt`的签名如下所示：

```
(defbolt name output-declaration *option-map & impl)
```

省略`option map`(选项映射)相当于具有`{: prepare false}`的`option map`(选项映射)。

## Simple bolts (简单 bolts)

我们从最简单的`defbolt`形式开始吧。这是一个将包含句子的元组分割成每个单词的元组的示例bolt：

```
(defbolt split-sentence ["word"] [tuple collector]
  (let [words (.split (.getString tuple 0) " ")]
    (doseq [w words]
      (emit-bolt! collector [w] :anchor tuple))
    (ack! collector tuple)
  ))
```

Since the option map is omitted, this is a non-prepared bolt. The DSL simply expects an implementation for the `execute` method of `IRichBolt`. The implementation takes two parameters, the tuple and the `outputCollector`, and is followed by the body of the `execute` function. The DSL automatically type-hints the parameters for you so you don't need to worry about reflection if you use Java interop. (由于感觉有不准确的地方，先留着方便优化。) 由于省略了option map(选项映射)，这是一个non-prepared bolt。DSL只是期望执行一个`IRichBolt`的`execute`方法。该实现需要两个参数，即`tuple`(元组)和`outputCollector`，后面是`execute`函数的正文。DSL会为你自动提示参数，所以如果您使用Java交互，不需要担心反射问题。

This implementation binds `split-sentence` to an actual `IRichBolt` object that you can use in topologies, like so: 此实现将`split-sentence`绑定到一个可用于topologies实现的`IRichBolt`对象，如下所示： `clojure (bolt-spec {"1" :shuffle} split-sentence :p 5)`

## Parameterized bolts (参数化 bolts)

有时候你想用其他参数来参数化你的bolts。例如，假设你想有一个可以接收到每个输入字符串后缀的bolts，并且希望在运行时设置该后缀。你可以在`defbolt`中通过在option map(选项映射)中包含：`:params`选项来执行此操作，如下所示：

```
(defbolt suffix-appender ["word"] {:params [suffix]}
  [tuple collector]
  (emit-bolt! collector [(str (.getString tuple 0) suffix)] :anchor tuple)
)
```

与前面的示例不同，`suffix-appender`将绑定到一个返回`IRichBolt`而不是直接作为`IRichBolt`对象的函数。这是通过在其option map(选项映射)中指定`:params`引起的。因此，在topology中使用`suffix-appender`，您可以执行以下操作：

```
(bolt-spec {"1" :shuffle}
           (suffix-appender "-suffix")
           :p 10)
```

## Prepared bolts (准备 bolts)

要做更复杂的bolts，如加入和流聚合的bolt，bolt需要存储状态。您可以通过在option map(选项映射)中创建一个通过包含`{:prepare true}`指定的prepared bolt 来实现此目的。例如，思考下这个实现单词计数的bolt:

```
(defbolt word-count ["word" "count"] {:prepare true}
  [conf context collector]
  (let [counts (atom {})]
    (bolt
      (execute [tuple]
        (let [word (.getString tuple 0)]
          (swap! counts (partial merge-with +) {word 1})
          (emit-bolt! collector [word (@counts word)] :anchor tuple)
          (ack! collector tuple)
        )))))

```

prepared bolt的实现是通过一个函数，它将topology的配置“TopologyContext”和“OutputCollector”作为输入，并返回“IBolt”接口的一个实现。此设计允许您围绕`execute`和`cleanup`的实现时进行闭包。

在这个例子中，单词计数存储在一个名为`counts`的映射的闭包中。`bolt`宏用于创建`IBolt`实现。`bolt`宏是一种比简化实现界面更简洁的方法，它会自动提示所有的方法参数。该`bolt`实现了更新映射中的计数并发出新的单词计数的执行方法。

请注意，`prepared bolts` 中的`execute`方法只能作为元组的输入，因为`OutputCollector`已经在函数的闭包中（对于简单的bolts，`collector`是`execute`函数的第二个参数）。

Prepared bolts 可以像 simple bolts 一样进行参数化。

## Output declarations (输出声明)

Clojure DSL具有用于bolt输出的简明语法。声明输出的最通用的方法就是从stream id到stream spec的映射。例如：

```
{"1" ["field1" "field2"]
 "2" (direct-stream ["f1" "f2" "f3"])
 "3" ["f1"]}
```

stream id 是一个字符串，而stream spec(流规范)是个字段的向量或由`direct-stream`包装的字段的向量。`direct stream`将流标记为`direct stream`（有关直接流的更多详细信息，请参阅[Concepts](#) 和[Direct groupings](#)）。

如果bolt只有一个输出流，您可以使用向量而不用输出声明的映射来定义bolt的默认流。例如：

```
["word" "count"]
```

这段bolt输出的声明 为默认 stream id 上的字段[“word” “count”]。

## Emitting, acking, and failing (发射, 确认和失败)

DSL可以使用OutputCollector: `emit-bolt!`, `emit-direct-bolt!`, `ack!` 和 `fail!`, 而不用直接在OutputCollector上使用Java方法.

1. `emit-bolt!`: 将“OutputCollector”，发出的值（一个Clojure sequence）和`: anchor`以及`: stream`的关键字参数作为参数。`: anchor`可以是个single tuple或一个list of tuples, `: stream`是要发送到的流的id。若省略关键字参数则默认流会发出一个unanchored tuple。
2. `emit-direct-bolt!`: 将OutputCollector作为参数，发送元组的任务id，发送的值，以及把`: anchor`和`: stream`的关键字参数作为参数。此函数只能发出声明为direct streams的流。
3. `ack!`: 将“OutputCollector”作为元组确认参数。
4. `fail!`: 将“OutputCollector”作为元组失败参数

有关确认和锚定的更多信息，请参阅[保证消息处理](#)。

## defspout

`defspout`用于定义Clojure中的喷口。像螺栓一样，喷口必须是可序列化的，所以您不能只是在“Clojure”中引用“IRichSpout”来执行喷口实现。`defspout`围绕这个限制，为定义spouts提供了一个更好的语法，而不仅仅是实现一个Java接口。

`defspout`的签名如下：

`(defspout name output-declaration *option-map & impl)`

如果你省略选项映射，则默认为`{: prepare true}`。`defspout`的输出声明与`defbolt`语法相同。

这里有个实现`defspout`的一个例子[storm-starter](#):

```
(defspout sentence-spout ["sentence"]
  [conf context collector]
  (let [sentences ["a little brown dog"
                  "the man petted the dog"
                  "four score and seven years ago"
                  "an apple a day keeps the doctor away"]]
    (spout
      (nextTuple []
        (Thread/sleep 100)
        (emit-spout! collector [(rand-nth sentences)]))
      )
    (ack [id]
```

```
;; You only need to define this method for reliable spouts
;; (such as one that reads off of a queue like Kestrel)
;; This is an unreliable spout, so it does nothing here
))))
```

该实现将topology配置的“TopologyContext”和“SpoutOutputCollector”作为输入。该实现返回一个ISpout对象。这里，nextTuple函数从sentence发出一个随机语句。

这个spout不是可靠的，所以ack和fail方法永远不会被调用。一个可靠的端口将在发出元组时添加一条消息ID，然后当元组完成或失败时，将会调用ack或fail。有关Storm中可靠性如何工作的更多信息，请参阅[保证消息处理](#)。

emit-spout! 将“SpoutOutputCollector”和新元组的参数作为参数发送，并接受作为关键字参数：stream和：id。: stream为指定要发送的流，: id为指定元组的消息ID（在ack'和fail回调中使用）。省略这些参数会为默认输出流发出一个unanchored tuple。

这还有一个emit-direct-spout !函数，他会发出一个direct stream的元组，并附加一个任务id作为的第二个参数来发送这个元组。

Spouts可以像bolts一样进行参数化，在这种情况下，symbol绑定到返回“IRichSpout”的函数而不是“IRichSpout”本身。您还可以声明一个unprepared spout，它只定义nextTuple方法。以下是在运行时发出随机语句参数化的unprepared spout示例：

```
(defspout sentence-spout-parameterized ["word"] {:params [sentences] :prepare false}
[collector]
(Thread/sleep 500)
(emit-spout! collector [(rand-nth sentences)]))
```

以下示例说明了如何在spout-spec中使用此spout：

```
clojure (spout-spec (sentence-spout-parameterized ["the cat jumped over the door" "greetings from a ..
```

## Running topologies in local mode or on a cluster (在本地模式或集群上运行topologies)

要想使用远程模式或本地模式提交topologies，只需像Java一样使用“StormSubmitter”或“LocalCluster”类。这就是Clojure DSL。

要创建topology配置，最简单的方法是使用org.apache.storm.config命名空间来定义所有可能配置的常量。常量与“Config”类中的静态常量相同，但是使用的是破折号而不是下划线。例如，这有一个topology配置，将workers数设置为15，并以调试模式配置topology：

```
{TOPOLOGY-DEBUG true  
TOPOLOGY-WORKERS 15}
```

## Testing topologies (测试topologies)

关于测试Clojure中的topologies，[博文](#)及其[后续](#)很好地概述了Storm的强大内置功能。

# 使用没有jvm的语言编辑storm

- 两部分：创建topologies 以及 使用其他语言来实现 spouts 和bolts
- 用另一种语言创建topologies 是比较容易的，因为topologies 用的是thrift 的结构
- 用另一种语言实现 spouts 和 bolts 被称为“multilang components ”或“shelling ”
  - 以下是协议的规范：[Multilang协议](#)
  - thrift 结构允许你将多个组件明确定义为程序和脚本（例如，使用python编写你的bolt的文件）
  - 在Java中，您可以通过重写ShellBolt或ShellSpout来创建multilang组件
    - 请注意，输出字段声明发生在thrift 结构中，所以在java中创建multilang 组件需要按照以下方式：
      - 在java中声明字段，通过在shellbolt的构造函数中指定它来处理另一种语言的代码
    - multilang使用stdin / stdout上的json消息与子进程进行通信
    - storm 带有Ruby、Python和实现协议的奇特适配器。下面展示一个python的示例 - python支持emitting, anchoring, acking, 以及 logging
- “storm shell ”命令使得构建jar和上传到nimbus变得更加容易 - 创建jar以及上传它
  - 使用主机/端口nimbus和jarfile id来调用你的程序

## 关于在非JVM语言中实现DSL的注意事项

正确的打开方式地方是src / storm.thrift。由于storm topologies 是Thrift结构，Nimbus是Thrift守护进程，您可以使用任何语言创建和提交topologies 。

当您为spouts 和bolts 创建Thrift结构体时，将在ComponentObject结构体中指定spout 或bolt的代码：

```
union ComponentObject {  
    1: binary serialized_java;  
    2: ShellComponent shell;  
    3: JavaObject java_object;  
}
```

对于非JVM DSL，您需要使用“2”和“3”。 ShellComponent允许您指定运行该组件的脚本（例如，您的python代码）。而JavaObject允许您为组件指定本地java的spout 和bolt （Storm将使用反射来创建该spout 或bolt ）。

有一个“storm shell ”命令有助于提交topology 。它的用法是这样的：

```
storm shell resources/ python topology.py arg1 arg2
```

storm shell 会 resources/ 打成一个jar ,并上传这个jar到Nimbus , 并像下面这样调用你的 topology.py脚本:

```
python topology.py arg1 arg2 {nimbus-host} {nimbus-port} {uploaded-jar-location}
```

之后你可以使用Thrift API连接到Nimbus, 并提交topology , 将{uploaded-jar-location}传递到submitTopology方法。为了方便参考我在下面展示了submitTopology类的定义。

```
void submitTopology(1: string name, 2: string uploadedJarLocation, 3: string jsonConf, 4: StormTopo  
throws (1: AlreadyAliveException e, 2: InvalidTopologyException ite);
```

# Distributed RPC

distributed RPC（分布式RPC）(DRPC) 的设计目的是充分利用Storm的计算能力实现高密度的并行实时计算。Storm topology（拓扑）接受若干个函数参数作为输入，然后输出这些函数调用的结果。

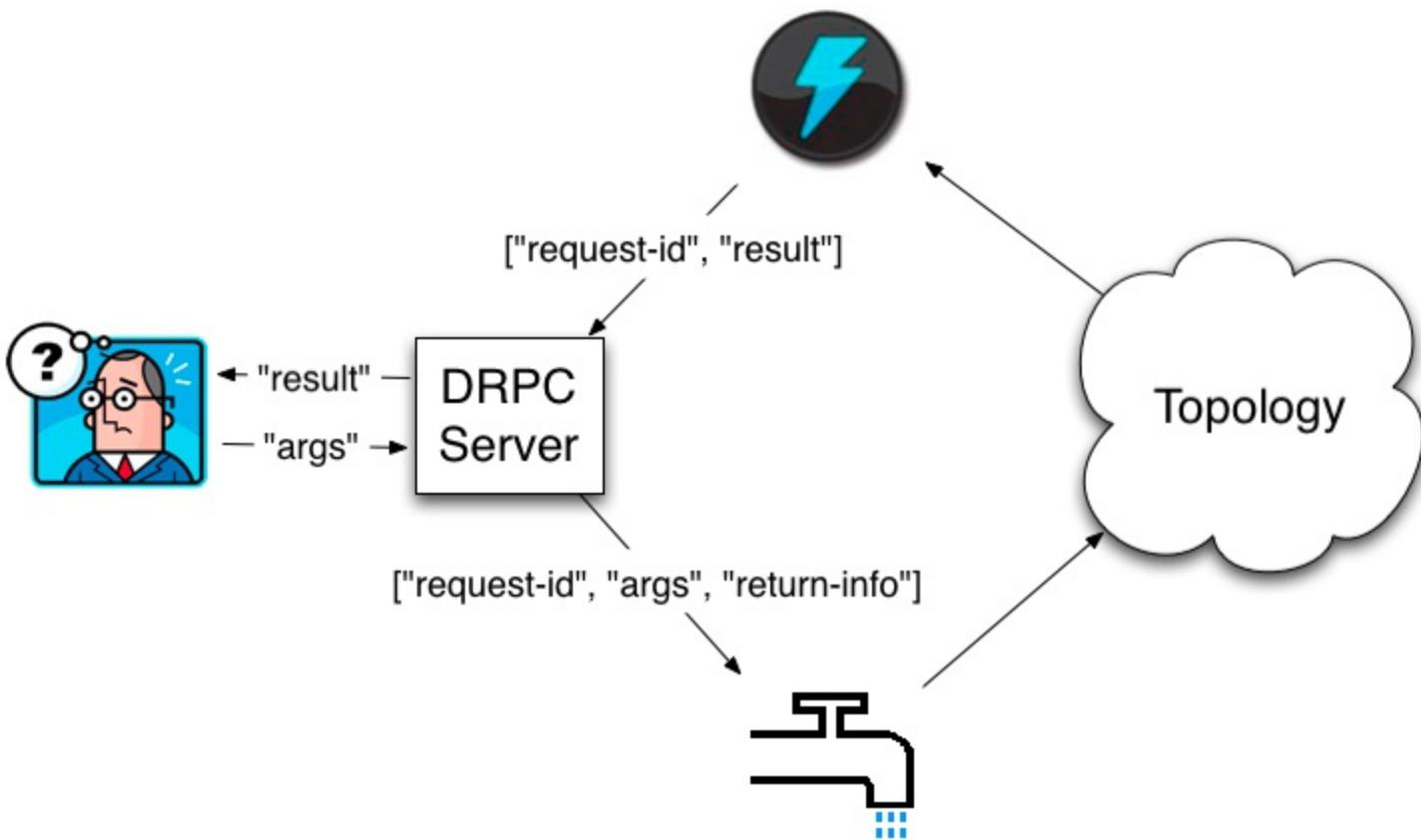
严格来说，DRPC不能够算作Storm的一个特性，因为它是一种基于Storm 原语（Stream, Spout, Bolt, Topology）实现的设计模式。DRPC可以脱离Storm，打包出来作为一个独立的库，但是它和Storm集成在一起更有用。

## High level 概述

Distributed RPC通过 "DRPC sever"（Storm包含了这个实现）来进行协调。DRPC server 负责接收 RPC 请求，发送请求到 Storm 对应的 topology（拓扑），再从 Storm topology（拓扑）上得到结果，最后发送给等待的客户端。从客户端的角度来看，DRPC调用和普通的RPC调用没有什么区别。例如，以下是一个使用参数 "<http://twitter.com>"，调用 "reach" 函数计算结果的例子：

```
DRPCCClient client = new DRPCCClient("drpc-host", 3772);
String result = client.execute("reach", "http://twitter.com");
```

Distributed RPC 工作流如下：



客户端发送要调用的函数名称和函数所需参数到 DRPC server。实现该函数的 topology（拓扑）使用一个 DRPCSpout 从 DRPC server 接收一个 function invocation stream（函数调用流）。DRPC sever 每一个函数调用都会给予一个唯一性的 id， topology（拓扑）计算完结果，使用一个叫做 `ReturnResults` 的 bolt 连接到DRPC server，根据函数调用的 id 将结果返回。DRPC sever 使用 id 来匹配client等待的是哪个结果，unblock 等待的client，将结果返回。

## LinearDRPCTopologyBuilder

Storm 有一个 topology（拓扑） 构造器叫 `LinearDRPCTopologyBuilder`，可以自动化 DRPC 所涉及的几乎所有步骤，这些包括：

1. 建立 spout
2. 返回结果到 DRPC server
3. 给 bolts 提供聚合 tuples 的功能

我们一起来看一个简单的例子。下面实现了一个DRPC topology（拓扑），返回输入参数添加一个"!"。

```
public static class ExclaimBolt extends BaseBasicBolt {
    public void execute(Tuple tuple, BasicOutputCollector collector) {
        String input = tuple.getString(1);
```

```

        collector.emit(new Values(tuple.getValue(0), input + "!"));

    }

    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);
    // ...
}

```

正如你所看到的，我们需要做的事情非常少。当创建 `LinearDRPCTopologyBuilder` 的时候，你告诉它相应**topology**（拓扑）的 **DRPC** 函数名称。一个单一的 **DRPC sever** 可以协调很多函数，函数与函数之间使用函数名称进行区分。你声明的第一个**bolt** 会得到一个二维的 **tuple**，**tuple**的第一个参数是 **request id**，第二个字段是请求的参数。

`LinearDRPCTopologyBuilder` 希望最后一个 **bolt** 会输出一个二维的[id, result]格式的输出流。最后，所有中间结果产生的 **tuples** 必须包含 **request id** 作为第一个字段。

在这个例子中，`ExclaimBolt` 简单的在 **tuple** 的第二个 **field** 后添加了一个 "!" 符合。`LinearDRPCTopologyBuilder` 帮我们协调处理了这些事情：连接 **DRPC server**，并将结果返回。

## DRPC 本地模式

**DRPC** 可以以本地模式运行，下面是以本地模式运行上面例子的代码：

```

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了一个 **DRPC server**，就像 `LocalCluster` 在进程内模拟一个**storm**集群。然后创建 `LocalCluster` 对象在本地模式运行 **topology**（拓扑）。`LinearDRPCTopologyBuilder` 有单独的方法创建本地 **topology**（拓扑）和远程的**topolgy**（拓扑）。在本地模式里面，`LocalDRPC` 对象不会和任何端口绑定，所以 **topology**（拓扑）需要知道对象和哪个 **DRPC** 交互。这就是为什么 `createLocalTopology` 需要一

个 LocalDRPC 作为输入。

启动 topology (拓扑) 后，你可以使用 LocalDRPC 的 execute 方法做 DRPC 调用。

## DRPC远程模式

在一个真实集群上面使用DRPC也是非常简单的，有三个步骤：

1. 启动 DRPC servers
2. 配置 DRPC servers 的地址
3. 提交 DRPC topologies (DRPC 拓扑) 到Storm集群。

可以使用 storm 脚本启动一个DRPC server，就像启动 Nimbus 和 UI 节点一样。

```
bin/storm drpc
```

下一步，你可以配置 Storm 集群知道 DRPC sever(s) 的地址。DRPCSpout 需要知道这个 DRPC 地址，从而得到函数调用的请求。这个可以配置 storm.yaml 文件或者通过代码配置在 topology (拓扑) 参数中。通过 storm.yaml 配置像下面这样：

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

最后，你使用 StormSubmitter 运行 DRPC topologies (拓扑)，和你提交其他 topologies 一样。如果要以远程调用的方式运行上面的例子，用下面的代码：

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

createRemoteTopology 是用来创建符合 Storm cluster 的 topologies (拓扑) .

一个更复杂的例子

上面的例子只是用来说明 DRPC 的概念。我们来看一个更复杂的例子，需要使用 Storm 并行计算的 DRPC 功能。我们将看到这个例子是用来计算 Twitter 上每个URL的访问量 (reach) 。

一个URL的访问量 (reach) 是每个在Twitter上的URL暴露给不同的人的数量。为了计算访问量，你需要：

1. 得到所有 tweet 这个 URL 的人。

2. 得到步骤1中所有人的粉丝
3. 对所有粉丝进行去重
4. 对步骤3的粉丝求和。

单个 URL 的访问量计算会涉及成千上万次数据库调用以及数以百万的粉丝记录。这是一个很大很大的计算量。正如你即将看到的，在 Storm 上实现这个功能是很简单的。在单机上，访问量可能需要几分钟才能完成；在 Storm 集群上，即使是很难计算的 URLs 也会在几秒内计算出访问量。

一个访问量的 topology 的例子定义在 `storm-starter`: [这里](#)。下面是你如何定义这个访问量的topology。

```
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topolgy（拓扑）执行有四个步骤：

1. `GetTweeters` 得到tweet指定URL的用户列表。这个Bolt将输入流 `[id, url]` 转换成输出流 `[id, tweeter]`。每个 `url` tuple 被映射成多个 `tweeter` tuples。
2. `GetFollowers` 得到步骤1 tweeters的followers。该Bolt将输入流 `[id, tweeter]` 转换成 `[id, follower]`。在所有的任务中，当有人是多个 tweet 相同 url 的粉丝，那么 follower 的 tuple就会重复。
3. `PartialUniquer` 按照follower id 对followers流进行分组。相同的follower进入同一个任务。所以 `PartialUniquer` 的每个任务会收到完全独立的followers结果集。一旦 `PartialUniquer` 接受到针对request id 的所有 follower tuple，就会发出它 followers 子集的计数。
4. 最后， `CountAggregator` 获取到每一个 `PartialUniquer` 任务中接受到部分结果，累加结果后计算出访问量。

我们来看一下 `PartialUniquer` bolt:

```
public class PartialUniquer extends BaseBatchBolt {
    BatchOutputCollector _collector;
    Object _id;
    Set<String> _followers = new HashSet<String>();
```

```

@Override
public void prepare(Map conf, TopologyContext context, BatchOutputCollector collector, Object i
    _collector = collector;
    _id = id;
}

@Override
public void execute(Tuple tuple) {
    _followers.add(tuple.getString(1));
}

@Override
public void finishBatch() {
    _collector.emit(new Values(_id, _followers.size()));
}

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

```

`PartialUniquer` 继承 `BaseBatchBolt` 实现了 `IBatchBolt` 接口。一个 batch bolt 提供了将一批 tuples 作为整体进行处理的 API。每一个 request id 都会创建一个新的 batch bolt 实例，Storm 会在适合时候负责清理实例。

当 `PartialUniquer` 在 `execute` 方法中接受到一个 follower tuple，将 follower 添加到 request id 对应的 `HashSet`。

Batch bolt 提供了 `finishBatch` 方法，当这个任务的 batch 所有的 tuples 处理完后进行调用。这次调用中，`PartialUniquer` 发送一个 tuple，包含 followers id 去重后的数量。

在内部实现上，`CoordinatedBolt` 用于检测指定的 bolt 是否已经收到指定 request id 的所有 tuples 元组。`CoordinatedBolt` 使用 direct streams 管理这个协调过程。

topology 的其他部分是容易理解。正如你看到，访问量计算的每一个步骤都是并行的，通过 DRPC 实现也是非常容易的。

## Non-linear DRPC topologies（拓扑）

`LinearDRPCTopologyBuilder` 只处理 "linear" DRPC 拓扑，计算过程可以像计算访问量一样分解成一系列步骤。不难想象，这需要一个更加复杂的 topology（拓扑），带有分支和合并的 bolts。现在，要完成这种计算，你需要放弃直接使用 `CoordinatedBolt`。请务必在邮件列表中讨论关于 non-linear DRPC topologies（拓扑）的应用场景，以便为 DRPC topologies（拓

扑) 提供更一般的抽象。

**LinearDRPCTopologyBuilder** 是如何运行的。

- DRPCSpout 发出 [args, return-info], 其中 return-info 包含 DRPC Server 的主机和端口号, 以及 DRPC Server 为该次请求生成的唯一id号
- 构造一个 Storm 拓扑包含以下部分:
  - DRPCSpout
  - PrepareRequest(生成一个请求id, 为return info创建一个流, 为args创建一个流)
  - CoordinatedBolt wrappers 和 direct groupings
  - JoinResult(将结果与return info拼接起来)
  - ReturnResult(连接到DRPC Server, 返回结果)
- LinearDRPCTopologyBuilder 是建立在 Storm 基本元素之上的高层抽象。

## Advanced

- KeyedFairBolt 用于组织同一时刻多个请求的处理过程
- 如何直接使用 CoordinatedBolt

# Transactional Topologies

请注意: Transactional topologies 已经摒弃 -- 使用 Trident 框架替代。

Storm guarantees data processing (保证数据处理) 至少一次。关于 Storm 问的最多的问题就是 "当 tuples 重发时, 你会如何做呢? 你会重复计算吗? "

Storm 0.7.0 版本介绍了 transactional topologies. 使得你可以在复杂的计算中做到 exactly once 的消息语义. 所以你可以以一种完全精准的, 可伸缩, 容错的方式执行程序。

和 Distributed RPC 一样, transactional topologies 并不是 Storm 的一种功能, 而是基于 Storm 原语 (streams, spouts, bolts, topologies) 构建的高级抽象。

这一页用来解释 transactional topology 抽象, 如何使用 API, 并提供 API 实现的细节。

## Concepts

我们一起来构建 transactional topologies (事务性拓扑) 的第一步. 我们先从简单的方法开始, 不断的完善达到我们想要的设计。

### Design 1

transactional topologies (事务性拓扑) 背后核心的思想就是对数据的处理提供严格的顺序性. 严格的顺序性就是说, 在处理 tuples 的时候, topology (拓扑) 将当前 tuple 成功处理完后才可以进行下一个 tuple 处理。

每个 tuple 都和一个 transaction id 关联. 当 tuple 失败需要重新处理的时候, tuple 会绑定相同的 transaction id 重新发送. tuple 的 transaction id 是自增长的, 所以第一个 tuple 的 transaction id 是 1, 第二个就是 2, 以此类推.

tuples 的严格顺序性使得你在 tuple 重新处理的时候可以保证 exactly-once 语义. 我们来看一个例子。

假设你想要计算 stream 中 tuples 的总数. 原来你可能只会将 count 总数存储在数据库中, 但是你现在将 count 总数和最新的 transaction id 存储在数据库中。在程序更新 db 中的 count 的时候, 只有当数据库中的 transaction id 和当前处理的 tuple 的 transaction id 不同的时候, 才会更新 count 总数. 考虑下面两种场景:

1. 数据库中的 *transaction id* 和当前 *transaction id* 不同: 因为 *transactions* (事务) 的严格顺序性, 我们可以确定当前的 *tuple* 并不代表 *count* 总数。所以我们安全的自增 *count*, 并更新 *transaction id*。
2. 数据库中的 *transaction id* 和当前 *transaction id* 相同: 那么我们知道这个 *tuple* 已经被并入计数, 可以跳过更新. 这个 *tuple* 一定在第一次更新数据库之后失败过, 在第二次处理成功后汇报之前.

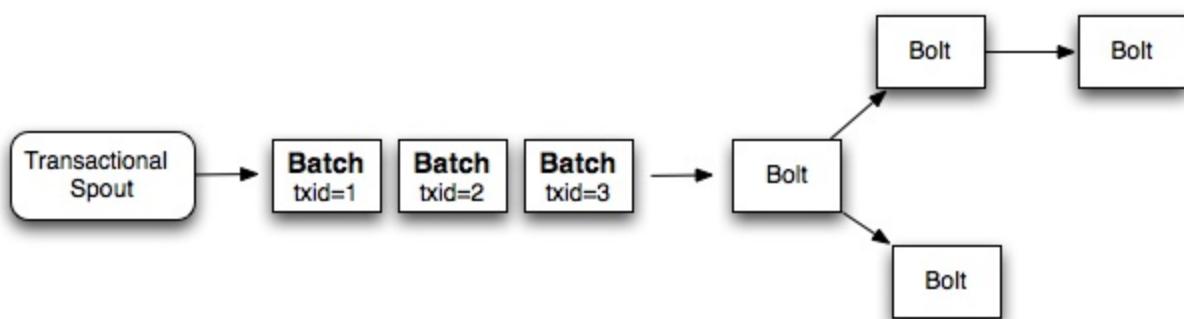
这种合理的和强一致性的事务保证了如果 *tuple* 失败了重新处理, 保存在数据库中的 *count* 也是准确的. 将 *transaction id* 存储到数据库和将 *value* 发送到 *kafka* 设计是一样的, 可以看 [this design document](#).

另外, *topology* 可以在相同的事务中的更新许多状态源, 并保证 *exactly-once* 语义. 如果有失败, 成功更新的会跳过重试, 失败更新的会进行重试. 例如, 你要处理 *tweeted urls* 的 *stream*, 你可以存储每一个 *url* 的 *tweet* 数量, 也可以存储每一个 *domain* (域名) 的 *tweet* 数量.

上面这种设计对在某一时刻处理一个 *tuple* 有一个比较严重的问题。必须等待每个 *tuple* 处理完成后, 才可以进行下一个处理, 这是非常低效的. 这种设计需要大量的数据库调用 (至少每个 *tuple* 一次), 这个设计很少用到并行, 所以它不是可扩展的.

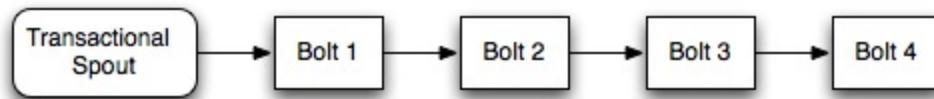
## Design 2

相对于一次只能处理一个 *tuple*, 更好的方式就是在每个 *transaction* (事务) 中批处理 *tuples*. 所以如果你要做一个全局的计数, 每次增加的是整个 *batch* 的数量. 如果 *batch* 失败了, 你需要重新处理这个失败的 *batch*. 相比于之前你要对每一个 *tuple* 分配一个 *transaction id*, 现在是对每个 *batch* 分配一个 *transaction id*. 并且处理 *batch* 也是严格有序的. 下面是这个设计的图表:



所以如果你每一个 batch 处理 1000 个 tuples, 相比于 design 1 (设计1) 你会少做  $1000 \times$  数量级的 数据库操作. 另外, 这种设计利用了 Storm 的并行计算特性, 每一个batch都可以并行计算.

虽然这种设计优于 design 1 (设计1), 但是它仍然不能有效的利用资源. topology 中的 workers 会花费大量的时间等待其他部分的计算完成。例如, 一个 topology (拓扑) 像下面这样:



当 bolt 1 完成部分处理后, 它所在的 worker 将是空闲的, 直到剩余的 bolt 完成后, 下一个 batch 才会从 spout 发送出来.

### Design 3 (Storm's design)

一个关键的实现就是并不是所有处理 batch 的工作都要严格有序。例如, 当计算一个全局计数, 需要计算两部分:

1. 计算每一个 batch 的局部 count
2. 通过局部 count 更新数据库里的全局 count

第二部分在计算 batch 过程中需要严格有序, 但是没有理由不并行计算第一部分。所以当 batch 1 正在更新数据库时, batch 2 到 10 可以计算他们的局部 count.

Storm 不同之处在于将 batch 计算分成两部分来完成:

1. 处理阶段: 这个阶段是可以并行处理 batches 的。
2. 提交阶段: 提交阶段, batches 是严格有序的. 所以 batch 2 必须等到 batch 1 提交成功后才可以进行提交.

这两个阶段合起来称之为“transaction”(事务)。许多 batch 在某一时间内处于处理阶段, 但只有一个 batch 处于提交阶段。如果处理阶段或者提交失败有失败的话, 将重新处理 batch (两个阶段都会重新处理).

### Design details

当使用 transactional topologies, Storm 为你提供以下信息:

- Manages state:** Storm 执行 transactional topologies 的时候，将所有的状态存储到 zookeeper.其中包括当前的 transaction id，也包括定义每个 batch 参数的 metadata 信息.
- Coordinates the transactions:** Storm 会管理一切必要的事情，来确定 transaction 什么时候处理或者提交.
- Fault detection:** Storm 利用 acking 框架来有效地确定批处理成功处理，成功提交或失败的时间。Storm 然后会适当地重新处理 batch 。你不必做任何暗示或 anchoring - Strom 管理所有这一切。
- First class batch processing API:** Storm 在常规螺栓之上层叠一个 API，以允许批量处理 tuples。Storm 管理所有协调，以确定任务何时已经接收到该特定事务的所有 tuples。Storm 也将照顾清理每笔交易的任何累计状态（如部分计数）。

最后需要注意的是，transactional topologies（事务拓扑）需要一个可以重播一批精确信息的 source queue. 像 Kestrel 是无法做到的. Apache Kafka 非常适合当这个 spout, , 而且 storm-kafka 包含一个用于 Kafka 的事务性 spout 实现.

## The basics through example

你通过使用 [TransactionalTopologyBuilder](#) 构建 transactional topologies .下面是一个 topology（拓扑）的 transactional topology 定义，用来计算输入 tuples 的总数. 代码来自于 storm-starter 的[TransactionalGlobalCount](#) .

```
MemoryTransactionalSpout spout = new MemoryTransactionalSpout(DATA, new Fields("word"), PARTITION_T
TransactionalTopologyBuilder builder = new TransactionalTopologyBuilder("global-count", "spout", sp
builder.setBolt("partial-count", new BatchCount(), 5)
    .shuffleGrouping("spout");
builder.setBolt("sum", new UpdateGlobalCount())
    .globalGrouping("partial-count");
```

[TransactionalTopologyBuilder](#) 将构造函数的输入作为 transactional topology 的 id，还有 topology 内的 spout id，一个事务性的 spout，还有事务性 spout 的并行度.transactional topology 的 id 是用来在Zookeeper 中存储 topology 的处理状态用的，以便如果重新启动 topology 后，将从停止的地方继续运行.

transactional topology 有一个 [TransactionalSpout](#)， [TransactionalSpout](#)

在 [TransactionalTopologyBuilder](#) 构造器中定义.在这个例子中， [MemoryTransactionalSpout](#) 用于从内存中分区的数据源（DATA 变量）中读取数据。第二个参数定义数据的字段，第三个参数指定

了每批 tuples 发出的 tuple 最大数量. 用于定义自己的 transactional spouts 将在本教程后面讨论。

然后就是 bolts, 这个 topology (拓扑) 并行计划全局 count. 第一个 Bolt BatchCount 使用 shuffle grouping 随机分割 input stream。第二个 Bolt UpdateGlobalCount 使用 global grouping, 并将局部 count 相加来获取 batch count. 如果有需要, 它会更新数据库中的全局 count.

下面是 BatchCount 的定义:

```
public static class BatchCount extends BaseBatchBolt {
    Object _id;
    BatchOutputCollector _collector;

    int _count = 0;

    @Override
    public void prepare(Map conf, TopologyContext context, BatchOutputCollector collector, Object id) {
        _collector = collector;
        _id = id;
    }

    @Override
    public void execute(Tuple tuple) {
        _count++;
    }

    @Override
    public void finishBatch() {
        _collector.emit(new Values(_id, _count));
    }

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

BatchCount 在每一个 batch 被处理的时候, 都会实例化. 真正运行 bolt 的是 [BatchBoltExecutor](#), 用来管理这些对象的创建和清理.

`prepare` 方法使用 Storm config, topology 上下文, output collector, 这个批次 tuples 的 id 来进行参数设置。在 [transactional topologies](#) (事务拓扑) 中, id 将是一个 [TransactionAttempt](#) 对象。这个 batch bolt 的后巷可以在 [distributed rpc](#) 中使用, 也可以使用不同类型的 id. [BatchBolt](#) 也可以使用 id 的类型配置参数, 因此如果你打算在 [transactional topologies](#) (事务拓扑) 中使用 batch bolt, 你可以继承 [BaseTransactionalBolt](#):

```
public abstract class BaseTransactionalBolt extends BaseBatchBolt<TransactionAttempt> {  
}
```

所有在transactional topology (事务拓扑) 中发送的 tuples 必须让 TransactionAttempt 作为第一个字段，这可以让Storm 知道 tuple 属于哪个 batch。所以当你发送 tuple 的时候，必须保证这个要求。

TransactionAttempt 包含两个值： "transaction id" 和 "attempt id"。 "transaction id" 是 batch 的唯一性标识，同一个 batch 无论重复多少次处理，都不会改变。 "attempt id" 是 batch 中 tuple 的唯一标识，Storm 用来区分相同 batch 中不同的 tuples。没有 attempt id，Storm 可能会从 batch 发送之前开始重新处理。这是很可怕的。

每个 batch 发送的时候，transaction id 都加1。所以，第一个 batch 的 id 是 1，第二个就是 2，以此类推。

batch 中的每个 tuple 都会调用 execute 方法。在每次调用这个方法的时候，你应该在本地实例变量中累计 batch 的状态。BatchCount bolt 通过本地 counter 对每个 tuple 自增。

最后，当任务接受到指定的 batch 的所有 tuples 时，会调用 finishBatch 方法。当调用此方法时，BatchCount 会向 output stream 发出局部的 count。

下面是 UpdateGlobalCount 的定义：

```
public static class UpdateGlobalCount extends BaseTransactionalBolt implements ICommitter {  
    TransactionAttempt _attempt;  
    BatchOutputCollector _collector;  
  
    int _sum = 0;  
  
    @Override  
    public void prepare(Map conf, TopologyContext context, BatchOutputCollector collector, TransactionAttempt attempt) {  
        _collector = collector;  
        _attempt = attempt;  
    }  
  
    @Override  
    public void execute(Tuple tuple) {  
        _sum += tuple.getInteger(1);  
    }  
  
    @Override  
    public void finishBatch() {  
        Value val = DATABASE.get(GLOBAL_COUNT_KEY);  
        Value newval;  
        if(val == null || !val.txid.equals(_attempt.getTransactionId())) {  
            newval = new Value();  
            newval.txid = _attempt.getTransactionId();  
            DATABASE.put(GLOBAL_COUNT_KEY, newval);  
        } else {  
            newval = new Value();  
            newval.txid = _attempt.getTransactionId();  
            newval.integer += _sum;  
            DATABASE.put(GLOBAL_COUNT_KEY, newval);  
        }  
    }  
}
```

```

        if(val==null) {
            newval.count = _sum;
        } else {
            newval.count = _sum + val.count;
        }
        DATABASE.put(GLOBAL_COUNT_KEY, newval);
    } else {
        newval = val;
    }
    _collector.emit(new Values(_attempt, newval.count));
}

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

```

`UpdateGlobalCount` 对于 `transactional topologies` 是特殊的，所以它继承`BaseTransactionalBolt`类.在 `execute` 方法中，`UpdateGlobalCount` 通过将局部 batch累加在一起得到此 batch 的count.有趣的事情发生在 `finishBatch` 方法中.

首先，你会看到这个 Bolt 实现了 `ICommitter` 接口.这就告诉Storm `finishBatch` 方法是事务提交阶段的一部分.所以调用 `finishBatch` 将会按照 `transaction id` 严格有序（另一方面，`execute` 的调用可能发生在处理阶段或者提交阶段）。将 Bolt 标记为 `committer`的另外一种方式就是在 `TransactionalTopologyBuilder` 中使用`setCommitterBolt` 方法，而不是 `setBolt`。

`UpdateGlobalCount` 中的 `finishBatch` 的代码从数据库获取当前值，并将 `transaction id` 与此批次的 `transaction id` 进行比较。如果他们是一样的，它什么都不做。否则，数据库中的值就增加此batch 的局部 count。

在 `TransactionalWords` 类中的`storm-start`中可以找到更多涉及到更新多个数据库的 `transactional topology`示例.

## Transactional Topology API

本节概述了事务拓扑API的不同部分。

### Bolts

`transactional topology`（事务拓扑）中有三种 Bolt:

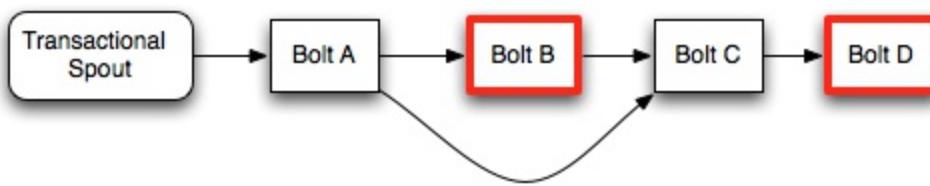
1. `BasicBolt`: 这个 Bolt 不处理 batches of tuples，只基于单个tuple输入发送tuples.
2. `BatchBolt`: 这个 Bolt 处理 batches of tuples,对于每个 tuple 调用`execute`，并在处理完

batch后调用 `finishBatch` 方法。

3. BatchBolt's that are marked as committers: 这个 Bolt 和常规的 `Batch Bolt` 之间的唯一区别是调用`finishBatch`时。Committer bolt 已经在提交阶段调用 `finishBatch` 方法。提交阶段只有在所有先前 batch 成功提交之后才能保证发生，并且将重新尝试，直到 topology (拓扑) 结构中的所有 Bolt 成功完成批处理的提交。有两种方式使 BatchBolt 成为 committer，通过使 `BatchBolt` 实现 `ICommitter` 标记接口，或者通过在 `TransactionalTopologyBuilder` 中使用 `setCommitterBolt` 方法。

## Processing phase vs. commit phase in bolts

为了确定 transaction (事务) 的处理阶段和提交阶段之间的差异，我们来看一个示例 topology (拓扑)：



在这种 topology (拓扑) 中，只有具有红色轮廓的 Bolts 才是 committers。

在处理阶段，Bolt A 将从 Spout 处理完整的 batch，调用 `finishBatch` 并将 tuples 发送到 Bolt B 和 C. Bolt B 是一个 committer，因此它将处理所有的 tuple，但是不会调用 `finishBatch`。Bolt C 也不会有 `finishBatch` 调用，因为它不知道它是否已经收到 Bolt B 的所有 tuple (因为 Bolt B 正在等待事务提交)。最后，Bolt D 将在其 `execute` 方法的调用期间接收 Bolt C 的 tuple.

当 batch 提交时，将在 Bolt B 上调用 `finishBatch`。一旦完成，Bolt C 现在可以检测到它已经接收到所有的 tuple，并将调用 `finishBatch`。最后，Bolt D 将收到完整的 batch 并调用 `finishBatch`。

请注意，即使 Bolt D 是 committer，它在收到整个 batch 时也不必等待第二个提交消息。由于它在提交阶段收到整个 batch，所以它将继续并完成 transaction (事务) 事务。

Committer bolts 在提交阶段就像 batch bolts 那样运行。committer bolts 和 batch bolts 之间的唯一区别是 committer bolts 在 transaction (拓扑) 的处理阶段不会调用 `finishBatch`。

## Acking

请注意，在使用 transactional topologies (事务拓扑) 时，您不必执行任何操作或

anchoring。Storm管理下面的所有这些。`acking` 策略被大量优化。

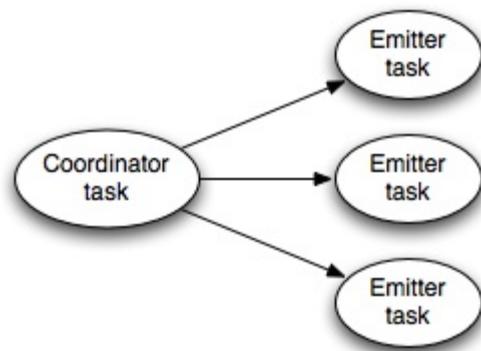
## Failing a transaction

当使用常规 bolts 时，可以在 `OutputCollector` 上调用 `fail` 方法来使该 tuple 的成员的 tuples tree 失败。由于 **transactional topologies** (事务拓扑) 隐藏了您的 `acking` 框架，因此它们提供了一种不同的机制来使 batch 失败（并导致 batch 被重播）。只是抛出一个 `FailedException`。与常规异常不同，这只会导致特定 batch 重播，并且不会使进程崩溃。

## Transactional spout

`TransactionalSpout` 接口与普通 `Spout` 接口完全不同。`TransactionalSpout` 实现发送批量的 tuples，并且必须确保为相同的事务 ID 始终发出同一批 tuples。

topology (拓扑) 拓扑正在执行时，`transactional spout` 看起来像这样：



左边的 `coordinator` (协调器) 是一个常规的 Storm spout，每当一个批处理被发送到一个事务中时，它会发出一个 tuple。emitters (发射器) 作为常规 Storm bolt 执行，并负责发射 batch 的实际 tuples。emitters (发射器) 使用 all grouping 订阅 coordinator (协调器) 的“batch emit” stream。

对于它发出的 tuple，需要是等幂的，需要一个 `TransactionalSpout` 来存储少量的状态。状态存储在 Zookeeper 中。

实现 `TransactionalSpout` 的细节在 [the Javadoc](#) 中。

## Partitioned Transactional Spout

一种常见的事务性出水口是从许多队列经纪人的一组分区中读取批次的。例如，这是 `TransactionalKafkaSpout` 的工作原理。`IPartitionedTransactionalSpout` 会自动执行管理每个分区的状态的记账工作，以确保幂等重播。有关详细信息。请参阅 [the Javadoc](#)

## Configuration

transactional topologies（事务性拓扑）有两个重要的配置位：

1. **Zookeeper**: 默认情况下，transactional topologies（事务拓扑）将在用于管理Storm集群的Zookeeper实例中存储状态。您可以使用“transactional.zookeeper.servers”和“transactional.zookeeper.port”配置覆盖此配置。
2. **Number of active batches permissible at once**: 您必须对可以一次处理的 batches 数设置限制。您可以使用“topology.max.spout.pending”配置进行配置。如果您没有设置此配置，它将默认为1。

## What if you can't emit the same batch of tuples for a given transaction id?

到目前为止，关于 transactional topologies（事务拓扑）的讨论假设您可以随时为相同的事务ID发出完全相同批次 tuple。那么如果不可能，你该怎么办？

考虑一下这个不可能的例子。假设您正在从分区消息代理读取 tuple（流在许多机器上分区），单个事务将包含所有单个机器的 tuple。现在假设其中一个节点在事务失败的同时下降。没有那个节点，就不可能重播刚刚为该事务ID播放的同一批 tuples。您的 topology（拓扑）拓扑中的处理将停止，因为它无法重播相同的批处理。唯一可能的解决方案是为该事务ID发出不同于之前发出的不同批处理。即使 batch 更改，仍然可以实现一次消息传递语义？

事实证明，您仍然可以使用非幂等的事务性端口在处理过程中实现完全一致的消息传递语义，尽管这在开发 topology（拓扑）中需要更多的工作。

如果 batch 可以更改给定的事务ID，那么我们迄今为止使用的逻辑“如果数据库中的事务ID与当前事务的id相同，则跳过更新”不再有效。这是因为当前批次与上次 transaction 提交的 batch 不同，因此结果不一定相同。您可以通过在数据库中存储更多的状态来解决此问题。我们再次使用在数据库中存储全局计数的示例，并假设批次的部分计数存储在partialCount 变量中。

而不是在数据库中存储一个如下所示的值：

```
class Value {  
    Object count;  
    BigInteger txid;  
}
```

对于非幂等事务端口，您应该存储一个如下所示的值：

```
class Value {  
    Object count;  
    BigInteger txid;  
    Object prevCount;  
}
```

更新的逻辑如下：

1. 如果当前 batch 的 transaction id 与数据库中的 transaction id 相同，请设置 `val.count = val.prevCount + partialCount`。
2. 否则，设置 `val.prevCount = val.count, val.count = val.count + partialCount` 和 `val.txid = batchTxid`。

这个逻辑是有效的，因为一旦你第一次提交一个特定的事务id，所有的事务id都不会再被提交。

transactional topologies（事务拓扑）有一些更细微的方面，使不透明的transactional spouts 口成为可能。

当 transaction 失败时，处理阶段中的所有后续 transaction 也被认为是失败的。这些 transactions 将被重新排放和再处理。没有这种行为，可能会发生以下情况：

1. Transaction A emits tuples 1-50
2. Transaction B emits tuples 51-100
3. Transaction A fails
4. Transaction A emits tuples 1-40
5. Transaction A commits
6. Transaction B commits
7. Transaction C emits tuples 101-150

在这种情况下，跳过 tuple 41-50。由于所有后续 transactions 失败，将会发生：

1. Transaction A emits tuples 1-50
2. Transaction B emits tuples 51-100
3. Transaction A fails (and causes Transaction B to fail)
4. Transaction A emits tuples 1-40
5. Transaction B emits tuples 41-90

6. Transaction A commits
7. Transaction B commits
8. Transaction C emits tuples 91-140

通过失败所有后续 transactions 失败，不会跳过 tuples。这也表明 transactions spout的要求是它们总是发出最后一个 transactions 处理的位置。

一个非幂等的 transactional spout 更简明地称为“不透明的投资点”（不透明与幂等相反）。  
[IOpaquePartitionedTransactionalSpout](#) 是一个用于实现不透明分区transactional spouts的接口，其中 [OpaqueTransactionalKafkaSpout](#) 是一个示例。只要您使用本节所述的更新策略，[OpaqueTransactionalKafkaSpout](#)可以承受丢失的单个Kafka节点，而不会牺牲精度。

## Implementation

transactional topologies（事务拓扑）的实现非常优雅。管理提交协议，检测故障和流水线批处理似乎很复杂，但一切事情都是对Storm 原语的简单映射。

数据流程如何工作：

transactional spout 是如何工作的

1. Transactional spout is a subtopology consisting of a coordinator spout and an emitter bolt
2. The coordinator is a regular spout with a parallelism of 1
3. The emitter is a bolt with a parallelism of P, connected to the coordinator's "batch" stream using an all grouping
4. When the coordinator determines it's time to enter the processing phase for a transaction, it emits a tuple containing the TransactionAttempt and the metadata for that transaction to the "batch" stream
5. Because of the all grouping, every single emitter task receives the notification that it's time to emit its portion of the tuples for that transaction attempt
6. Storm automatically manages the anchoring/acking necessary throughout the whole topology to determine when a transaction has completed the processing phase. The key here is that \*the root tuple was created by the coordinator, so the coordinator will receive an "ack" if the processing phase succeeds, and a "fail" if it doesn't succeed for any reason (failure or timeout).

7. If the processing phase succeeds, and all prior transactions have successfully committed, the coordinator emits a tuple containing the TransactionAttempt to the "commit" stream.
8. All committing bolts subscribe to the commit stream using an all grouping, so that they will all receive a notification when the commit happens.
9. Like the processing phase, the coordinator uses the acking framework to determine whether the commit phase succeeded or not. If it receives an "ack", it marks that transaction as complete in zookeeper.

更多概念:

- Transactional spouts are a sub-topology consisting of a spout and a bolt
  - the spout is the coordinator and contains a single task
  - the bolt is the emitter
  - the bolt subscribes to the coordinator with an all grouping
  - serialization of metadata is handled by kryo. kryo is initialized ONLY with the registrations defined in the component configuration for the transactionalspout
- the coordinator uses the acking framework to determine when a batch has been successfully processed, and then to determine when a batch has been successfully committed.
- state is stored in zookeeper using RotatingTransactionalState
- committing bolts subscribe to the coordinators commit stream using an all grouping
- CoordinatedBolt is used to detect when a bolt has received all the tuples for a particular batch.
  - this is the same abstraction that is used in DRPC
  - for committing bolts, it waits to receive a tuple from the coordinator's commit stream before calling finishbatch
  - so it can't call finishbatch until it's received all tuples from all subscribed components AND its received the commit stream tuple (for committers). this ensures that it can't prematurely call finishBatch

# Hooks

---

Storm 提供了一种hook机制，你可以用来实现在Storm各种事件上运行自定义的代码。通过继承 [BaseTaskHook](#) 类创建一个hook，并且可以覆盖你想要跟踪的事件的一些方法。

一共有两种方式注册你的hook:

1. 在 spout 的 `open` 方法或者 bolt 的 `prepare` 方法中使用 [TopologyContext](#) 方法.
2. 使用 Storm 配置表中的 "[topology.auto.task.hooks](#)" 配置项. 这些 hooks 会自动注册到每个 spout 和 bolt 中，这样就可以很方便地处理一些事情，例如集成自定义的监控系统.

# Storm Metrics

Storm 开放了一个 metrics 接口，用来汇报整个 topology 的汇总信息。 Storm 内部通过该接口可以跟踪各类统计数据： executor 和 acker 的数量；每个 bolt 的平均处理时延、 worker 节点的堆栈使用情况，这些信息都可以在 Nimbus 的 UI 界面中看到。

## Metric Types

Metrics 必须实现 [IMetric](#) 接口，IMetric 接口只包含一个方法 `getValueAndReset` -- 得到汇总值，并且重置为初始状态。例如，在 MeanReducer 中实现 running total/running count 的均值，然后两个值都重新设置为0.

Storm 提供了以下几种 metric 类型：

- [AssignableMetric](#) -- 将 metric 设置为指定值。此类型在两种情况下有用： 1. metric 本身为外部设置的值； 2. 你已经另外计算出了汇总的统计值。
- [CombinedMetric](#) -- 可以对 metric 进行关联更新的通用接口。
- [CountMetric](#) -- 返回 metric 的汇总结果。可以调用 `incr()` 方法来将加过自增； 调用 `incrBy(n)` 方法来将结果加上或者减去给定值。
  - [MultiCountMetric](#) -- 返回包含一组 CountMetric 的 HashMap
- [ReducedMetric](#)
  - [MeanReducer](#) -- 跟踪 `reduce()` 方法提供的运行状态均值结果（可以接受 `Double`、`Integer`、`Long` 等类型，内置的均值结果是 `Double` 类型）。 MeanReducer 确实是一个相当棒的家伙。
  - [MultiReducedMetric](#) -- 返回包含一组 ReducedMetric 的 HashMap

## Metrics Consumer

你可以监听和处理 topology (拓扑) 的metrics，通过注册 Metrics Consumer 到你的 topology.

注册 metrics consumer 到你的 topology，添加到你的 topology (拓扑) 配置，像下面这样：

```
conf.registerMetricsConsumer(org.apache.storm.metric.LoggingMetricsConsumer.class, 1);
```

你可以参考 [Config#registerMetricsConsumer](#)，然后根据 javadoc 覆盖 java 方法.

否则编辑 `storm.yaml` 配置文件:

```
topology.metrics.consumer.register:
  - class: "org.apache.storm.metric.LoggingMetricsConsumer"
    parallelism_hint: 1
  - class: "org.apache.storm.metric.HttpForwardingMetricsConsumer"
    parallelism_hint: 1
    argument: "http://example.com:8080/metrics/my-topology/"
```

注册 metrics consumer后，Storm 会添加 MetricsConsumerBolt 到你的 topology，每个 MetricsConsumerBolt 会从所有的 tasks 接受 metrics。相应的 MetricsConsumerBolt 的 parallelism 设置为 `parallelism_hint`，`component id` 设置为

`_metrics_<metrics consumer class name>`。如果你注册多次相同的类，`component id` 后会添加 `#<sequence number>`。

Storm 提供了一些内置的 metrics consumers，我们来看一下 topology（拓扑）中提供了哪些 metrics.

- `LoggingMetricsConsumer` -- 监听所有的 metrics，然后将数据dump 到日志文件(Tab Separated Values).
- `HttpForwardingMetricsConsumer` -- 监听所有的 metrics，并且将数据序列化，然后通过http post 到配置的url。Storm 提供 `HttpForwardingMetricsServer` 抽象类，你可以继承这个类，并且启动一个 HTTP sever，通过 `HttpForwardingMetricsConsumer` 处理发送的 metrics.

当然，Storm 开放了实现 Metrics Consumer 的 `IMetricsConsumer` 接口，你可以创建自定义 metrics consumers，绑定到相应的 topology（拓扑）。或者使用 Storm 社区内其他比较好的 Metrics Consumers 实现. 你可以参考 [versign/storm-graphite](#) 和 [storm-metrics-statsd](#).

当你实现自己的 metrics consumers，调用 `IMetricsConsumer#prepare`的时候，`argument` 需要传给 consumer 对象。所以你要参考 yaml 文件中配置的Java 类型，做好类型的分配。

请记住 MetricsConsumerBolt 只是 Bolt 类型的一种，所以 如果 metrics consumers 如果不能持续处理 metrics，topology 的吞吐量将会下降，所以你要和其他 Bolt 一样关注好 MetricsConsumerBolt。一个比较好的方式就是将 Metrics Consumers 设计为 `non-blocking`（非阻塞）的。

## 构建你自己的 metric (task level)

你可以通过注册 `IMetric` 到 Metric Registry（登记处），然后度量你的 metric. 假定你想要度

量 `Bolt#execute` 的执行次数。我们一起来定义这个 metric 实例。CountMetric 符合我们的应用场景。

```
private transient CountMetric countMetric;
```

你会发现，我们将CountMetric 定义为 transient。因为IMetric 并不是 Serializable 的，所以定义为 transient 可以避免很多问题。

下一步，我们初始化和注册 metric 实例。

```
@Override  
public void prepare(Map conf, TopologyContext context, OutputCollector collector) {  
    // other intialization here.  
    countMetric = new CountMetric();  
    context.registerMetric("execute_count", countMetric, 60);  
}
```

第一个和第二个参数很简单，metric 名称和 IMetric 实例。第三个参数[TopologyContext#registerMetric](#) 是发布和重置 metric 的间隔时间。

最后，当 Bolt.execute 执行的时候，自增countMetric的值。

```
public void execute(Tuple input) {  
    countMetric.incr();  
    // handle tuple here.  
}
```

请注意， topology 的sample rate 不适用于自定义 metrics，所以我们自己调用 incr() 方法。

`countMetric.getValueAndReset()` 每隔60秒都会被调用， ("execute\_count", value)也会被发送到 MetricsConsumer。

## Build your own metrics (worker level)

你可以注册 worker 级别的 metrics，将他们添加到集群所有的workers 的 `Config.WORKER_METRICS` 配置，或者所有workers上的指定 topology，通过 `Config.TOPOLOGY_WORKER_METRICS` 配置。

例如，我们可以添加 `worker.metrics` 配置到集群的 `storm.yaml`。

```
worker.metrics:  
  metricA: "aaa.bbb.cccddd.MetricA"  
  metricB: "aaa.bbb.cccddd.MetricB"
```

...

或者按照 `Map<String, String>` (`metric name`, `metric class name`) 格式, `key` 是 `Config.TOPOLOGY_WORKER_METRICS`, 来设置 config map。

`worker metrics` 实例有下面一些限制:

- A) `worker` 级别的 `metrics` 应该是一种 `gauge` 类, 因为它是从 `SystemBolt` 初始化和注册的, 不会暴露给 `user tasks`。
- B) `Metrics` 通过默认的构造器初始化, 并且不会对执行配置注入或者对象注入。
- C) `metrics Bucket size(secounds)` 已经修正为 `Config.TOPOLOGY_BUILTIN_METRICS_BUCKET_SIZE_SECS.`

## Builtin Metrics

Storm 的 [builtin metrics](#) 工具。

[`builtin\_metrics.clj`](#) 为内置的`metrics` 设置了数据结构, 其他框架组件可以使用 `facade` 方法来更新数据。`metrics` 在被调用的时候走计算逻辑-- 可以看例子 [`ack-spout-msg`](#) 的 `clj/b/s/daemon/daemon/executor.clj` 部分。

# Storm 状态管理

## 核心 Storm 中的状态支持

Storm 核心为 Bolt 提供用于保存和重新获取其操作状态的抽象. 提供一个基于内存的默认状态实现，同时还提供了一个使用 Redis 做状态保持的实现.

## 状态管理

若 Bolt 需要通过框架来管理和保持其状态，应该实现接口 `IStatefulBolt`, 或者继承类 `BaseStatefulBolt`, 然后实现方法 `void initState(T state)`. 方法 `initState` 在 Bolt 使用保存的历史状态进行初始化期间通过框架执行. 执行时机在 `prepare` 方法之后，在 Bolt 开始处理 Tuple 数据之前.

当前支持的唯一一种 `State` 实现是提供 key-value 映射的 `KeyValueState`.

例如，一个单词计数 bolt 可以使用 key-value 状态抽象实现单词计数，步骤如下.

1. 继承 `BaseStatefulBolt` 类，添加一个 `KeyValueState` 实例变量，用于存储单词到单词数量的映射.
2. 在 `init` 方法中用之前保存的状态来初始化 Bolt. 这里面含有上次程序运行的时候框架最后一次提交的单词计数.
3. 在 `execute` 方法中，更新单词计数.

```
public class WordCountBolt extends BaseStatefulBolt<KeyValueState<String, Long>> {  
    private KeyValueState<String, Long> wordCounts;  
    private OutputCollector collector;  
    ...  
    @Override  
    public void prepare(Map stormConf, TopologyContext context, OutputCollector collector) {  
        this.collector = collector;  
    }  
    @Override  
    public void initState(KeyValueState<String, Long> state) {  
        wordCounts = state;  
    }  
    @Override  
    public void execute(Tuple tuple) {  
        String word = tuple.getString(0);  
        Integer count = wordCounts.get(word, 0);  
        count++;  
        wordCounts.put(word, count);  
        collector.emit(tuple, new Values(word, count));  
        collector.ack(tuple);  
    }  
    ...
```

}

1. 框架周期性的检查并保存 Bolt 的状态 (默认每秒一次). 频率可以通过设置 storm config 的 `topology.state.checkpoint.interval.ms` 来自己定义。
2. 对于状态持久化, 可以设置 storm config 中的 `topology.state.provider` 来使用支持持久化的 state provider. 例如, 若使用基于 Redis 的 key-value 状态实现, 需要在 `storm.yaml` 文件中设置 `topology.state.provider: org.apache.storm.redis.state.RedisKeyValueStateProvider`. provider 实现代码的 jar 包需要放在 class path 下, 在这个例子中, 需要把 `storm-redis-*.jar` 置于 extlib 目录下.
3. state provider 的属性可以通过设置 `topology.state.provider.config` 来进行覆盖. 对于 Redis state, 是一个具有下列属性的 JSON 字符串.

```
{
  "keyClass": "Optional fully qualified class name of the Key type.",
  "valueClass": "Optional fully qualified class name of the Value type.",
  "keySerializerClass": "Optional Key serializer implementation class.",
  "valueSerializerClass": "Optional Value Serializer implementation class.",
  "jedisPoolConfig": {
    "host": "localhost",
    "port": 6379,
    "timeout": 2000,
    "database": 0,
    "password": "xyz"
  }
}
```

## 检查点机制

检查点通过一个内部的 `checkpoint spout` 来触发, 触发周期在

`topology.state.checkpoint.interval.ms` 指定. 如果在拓扑中至少有一个 `IStatefulBolt`, `topology builder` 会自动添加 `checkpoint spout`. 对于有状态的拓扑, `topology builder` 使用 `StatefulBoltExecutor` 包装 `IStatefulBolt`, 负责在收到 `checkpoint tuple` 的时候来执行状态提交. 无状态的 Bolt 被包装在 `CheckpointTupleForwarder`, 仅会转发 `checkpoint tuple` 以确保其可以贯穿整个拓扑DAG(有向无环图). `checkpoint tuple` 在一个名为 `$checkpoint` 的内部 stream 中流动. `topology builder` 组织 `checkpoint spout` 源流出的 `checkpoint stream` 穿过整个拓扑.



当到了检查周期, `checkpoint tuples` 被 `checkpoint spout` 发射出来. 一旦接收到 `state tuple`, `Bolt` 的状态就会被保存, 然后 `checkpoint tuple` 会转发到下一个组件. 每一个 `Bolt` 在保存状态之前, 会在所有的输入流上等待 `checkpoint` 到达, 使得状态表现为一个跨整个拓扑的持续的状态. 一旦 `checkpoint spout` 从所有的 `Bolt` 中接收到ACK消息, 状态提交就完成了, 事务会被 `checkpoint spout` 记录为已提交.

`checkpoint` 当前不会检查 `Spout` 的状态. 目前, 一旦所有的 `Bolt` 被检查完毕, 并且一旦 `checkpoint tuple` 被 ack, `Spout` 发射的 `tuples` 也会被 ack. 这也意味着,

`topology.state.checkpoint.interval.ms` 要小于 `topology.message.timeout.secs`.

状态提交的工作方式就像一个具有 `准备` 和 `提交` 阶段的三段式提交协议, 以达到跨整个拓扑的状态的保存操作具有一致性和原子性.

## 恢复

恢复阶段会在拓扑首次启动的时候触发. 如果前置事务没有成功装备好, 会向拓扑中发送一个 `rollback` 消息, `Bolt` 会丢弃已经就绪的事务. 如果前置事务成功准备好但是未提交, 会向拓扑中发送一个 `commit` 消息让所有已经就绪的事务可以被提交. 当这些步骤完成后, `Bolt` 状态初始化完成.

恢复也会在其中一个 `Bolt` 未成功确认 `checkpoint` 消息或者 `worker` 在这中间挂了的时候触发. 因此, 当 `supervisor` 重启一个 `worker`, `checkpoint` 机制会确保 `Bolt` 使用之前的状态初始化, 同时检查操作会从上次离开的点继续执行.

## 可靠性

Storm 使用 `acking` 机制在 `tuples` 处理失败的时候进行重新发送. 有可能状态已经提交但是 `worker` 在确认(ack) `tuple` 之前挂掉. 在这种情况下重新发送的 `tuple` 会导致状态重复更新. 当前, `StatefulBoltExecutor` 在接收到一个流中的 `checkpoint tuple` 以后继续从一个流中获取并处理 `tuple`, 同时等待 `checkpoint` 到达其他输入流以保存状态. 这也可能导致恢复期间造成重复的状态更新.

状态抽象并不能消除重复, 当前仅提供'至少一次'的保障.

为了提供'至少一次'的保障, 有状态拓扑中的所有 `Bolt` 都会对 `Tuple` 进行标记, 同时在处理完成后发射并确认输入 `Tuple`. 对于无状态的 `Bolt`, 继承 `BaseBasicBolt` 可以自动管理"标记/确认"操作. 有状态的 `Bolt` 标记 `Tuple` 同时在处理完成后发射和确认 `tuple`, 就像上面"状态管

理"一节中的 `WordCountBolt`.

## IStateful bolt 钩子

`IStateful` 接口提供钩子方法用以在有状态 `Bolt` 中可以实现一些自定义的动作

```
/**  
 * This is a hook for the component to perform some actions just before the  
 * framework commits its state.  
 */  
void preCommit(long txid);  
  
/**  
 * This is a hook for the component to perform some actions just before the  
 * framework prepares its state.  
 */  
void prePrepare(long txid);  
  
/**  
 * This is a hook for the component to perform some actions just before the  
 * framework rolls back the prepared state.  
 */  
void preRollback();
```

这个功能是可选的，并且有状态 `Bolt` 未提供任何实现。提供这个功能是为了可以在状态抽象的顶层(我们可能想在有状态 `Bolt` 的状态准备好之前做一些其他动作如提交或者回滚的地方)建立其他系统级组件。

## 提供自定义状态实现

当前唯一支持的 `State` 实现是提供 `key-value` 的映射的 `KeyValueState`。

自定义状态实现应当为接口 `org.apache.storm.State` 的方法提供实现。这些方法是 `void prepareCommit(long txid)`, `void commit(long txid)`, `rollback()`. `commit()` 方法是可选的且在 `Bolt` 管理自己的状态的时候非常有用。这些当前仅用于内部系统 `Bolt`, 例如 `CheckpointSpout` 在保存自己状态的时候。

`KeyValueState` 的实现也应当实现定义在接口 `org.apache.storm.state.KeyValueState` 中的方法。

## State provider

框架通过对的 `StateProvider` 来实例化状态。一个自定义的状态应当也提供一个可以加载和返回基于命名空间的状态的 `StateProvider` 实现。每一个状态属于一个独有的命名空间。命名空间通常是每个 `Task` 唯一的，因此每个任务可以有自己的状态。`StateProvider` 和相应的 `State` 实现应该位于 `Storm` 的 `class path` 下 (一般放在 `extlib` 目录中)。

# Windowing Support in Core Storm

Storm core 支持处理落在窗口内的一组元组。窗口操作指定了一下两个参数

1. 窗口的长度 - 窗口的长度或持续时间

2. 滑动间隔 - 窗口滑动的时间间隔

## 滑动窗口

元组被分组在窗口和每个滑动间隔窗口中。一个元组可以属于多个窗口。

例如一个持续时间长度为 10 秒和滑动间隔 5 秒的滑动窗口。

```
.....| e1 e2 | e3 e4 e5 e6 | e7 e8 e9 |... -5 0 5 10 15 -> time |<----- w1 -->| |<----- w2
```

窗口每5秒进行一次评估，第一个窗口中的某些元组与第二个窗口重叠。

注意：窗口第一次滑动在  $t = 5\text{s}$ ，并且将包含在前 5 秒钟内收到的事件。

## Tumbling Window

元组根据时间或数量被分组在一个窗口中。任何元组只属于其中一个窗口。

例如一个持续时间长度为 5s 的 tumbling window。

	e1 e2		e3 e4 e5 e6		e7 e8 e9	...
0	5	10	15	-> time		
w1	w2	w3				

窗口每五秒进行一次评估，并且没有窗口重叠。

Storm 支持指定窗口长度和滑动间隔作为元组数的计数或持续时间。 bolt 接口 `IWindowedBolt` 需要由窗口支持的bolts来实现。

```
public interface IWindowedBolt extends IComponent {  
    void prepare(Map stormConf, TopologyContext context, OutputCollector collector);  
    /**  
     * Process tuples falling within the window and optionally emit  
     * new tuples based on the tuples in the input window.  
     */  
    void execute(TupleWindow inputWindow);  
    void cleanup();  
}
```

每次窗口激活时，都会调用 `execute` 方法。`TupleWindow` 的参数允许访问窗口中的当前元组，过期的元组以及自上次窗口计算后添加的新元组，这对于高效的窗口计算将是有用的。

需要窗口支持的 Bolts 一般会扩展为 `BaseWindowedBolt`，它有用来指定窗口长度和滑动间隔的 apis.

例如

```
public class SlidingWindowBolt extends BaseWindowedBolt {  
    private OutputCollector collector;  
  
    @Override  
    public void prepare(Map stormConf, TopologyContext context, OutputCollector collector) {  
        this.collector = collector;  
    }  
  
    @Override  
    public void execute(TupleWindow inputWindow) {  
        for(Tuple tuple: inputWindow.get()) {  
            // do the windowing computation  
            ...  
        }  
        // emit the results  
        collector.emit(new Values(computedValue));  
    }  
}  
  
public static void main(String[] args) {  
    TopologyBuilder builder = new TopologyBuilder();  
    builder.setSpout("spout", new RandomSentenceSpout(), 1);  
    builder.setBolt("slidingwindowbolt",  
        new SlidingWindowBolt().withWindow(new Count(30), new Count(10)),  
        1).shuffleGrouping("spout");  
    Config conf = new Config();  
    conf.setDebug(true);  
    conf.setNumWorkers(1);  
  
    StormSubmitter.submitTopologyWithProgressBar(args[0], conf, builder.createTopology());  
}
```

支持以下窗口配置

`withWindow(Count windowLength, Count slidingInterval)`  
基于元组计数的滑动窗口，在多个tuples进行 `slidingInterval` 滑动之后。

`withWindow(Count windowLength)`  
基于元组计数的窗口，它与每个传入的元组一起滑动。

`withWindow(Count windowLength, Duration slidingInterval)`  
基于元组计数的滑动窗口，在`slidingInterval`持续时间滑动之后。

`withWindow(Duration windowLength, Duration slidingInterval)`  
基于持续时间的滑动窗口，在`slidingInterval`持续时间滑动之后。

`withWindow(Duration windowLength)`  
基于持续时间的窗口，它与每个传入的元组一起滑动。

```
withWindow(Duration windowLength, Count slidingInterval)
```

基于时间的滑动窗口配置在`slidingInterval`多个元组之后滑动。

```
withTumblingWindow(BaseWindowedBolt.Count count)
```

计数的tumbling窗口在指定的元组数之后tumbles.

```
withTumblingWindow(BaseWindowedBolt.Duration duration)
```

基于持续时间的tumbling窗口在指定的持续时间后tumbles.

## 元组时间戳和乱序元组

默认情况下，在窗口中追踪的时间戳是 bolt 处理元组的时间。窗口计算是根据正在处理的时间戳进行的。 Storm 支持基于源生成的时间戳的追踪窗口。

```
/**  
 * Specify a field in the tuple that represents the timestamp as a long value. If this  
 * field is not present in the incoming tuple, an {@link IllegalArgumentException} will be thrown.  
 *  
 * @param fieldName the name of the field that contains the timestamp  
 */  
public BaseWindowedBolt withTimestampField(String fieldName)
```

上述 `fieldName` 的值将从传入的元组中查找并考虑进行窗口计算。如果该元组中不存在该字段，将抛出异常。或者，[TimestampExtractor](#) 可以用于从元组导出时间戳值（例如，从元组中的嵌套字段提取时间戳）。

```
/**  
 * Specify the timestamp extractor implementation.  
 *  
 * @param timestampExtractor the {@link TimestampExtractor} implementation  
 */  
public BaseWindowedBolt withTimestampExtractor(TimestampExtractor timestampExtractor)
```

与时间戳字段 `name/extractor` 一起，可以指定一个时间滞后参数，它指示具有无序时间戳的元组的最大时间限制。

```
/**  
 * Specify the maximum time lag of the tuple timestamp in milliseconds. It means that the tuple time  
 * cannot be out of order by more than this amount.  
 *  
 * @param duration the max lag duration  
 */  
public BaseWindowedBolt withLag(Duration duration)
```

例如：如果滞后是5秒，并且元组 `t1` 到达时间戳为 `06: 00: 05` 没有元组可能会在早于 `06: 00: 00` 的元组时间戳到达。如果一个元组在 `t1` 之后到达时间戳 `05:59:59`，并且窗口已经移动过 `t1`

了，它将被视为迟到的元组。默认情况下不处理迟到的元组，只需在INFO级别打印到工作日志文件。

```
```java /**
 * Specify a stream id on which late tuples are going to be emitted.
 * They are going to be accessible via the
 * {@link org.apache.storm.topology.WindowedBoltExecutor#LATE_TUPLE_FIELD} field.
 * It must be defined on a per-component basis, and in conjunction with the
 * {@link BaseWindowedBolt#withTimestampField}, otherwise {@link IllegalArgumentException}
 * will be thrown.
 *
 * @param streamId the name of the stream used to emit late tuples on */
public BaseWindowedBolt withLateTupleStream(String streamId)
```

通过指定上述`streamId`来更改此行为。在这种情况下，迟到的元组将在指定的流中发出并通过`WindowedBoltEx`字段。

### ### Watermarks

为了处理具有时间戳字段的元组，storm根据传入的元组时间戳内部计算watermarks。Watermark是所有输入流中最定期的（默认每秒），watermark时间戳被发出，如果基于元组的时间戳被使用，这被认为是窗口计算的clock tick

```
```java
/**
 * Specify the watermark event generation interval. For tuple based timestamps, watermark events
 * are used to track the progress of time
 *
 * @param interval the interval at which watermark events are generated
 */
public BaseWindowedBolt withWatermarkInterval(Duration interval)
```

当接收到watermark时，将对所有时间戳记进行评估。

例如，考虑具有以下窗口参数基于元组的时间戳处理，

```
Window length = 20s, sliding interval = 10s, watermark emit frequency = 1s, max lag = 5s
```



当前 ts = 09:00:00

在09:00:00到09:00:01收到的元

组e1(6:00:03), e2(6:00:05), e3(6:00:07), e4(6:00:18), e5(6:00:26), e6(6:00:36)

在time t = 09:00:01, watermark w1 = 6:00:31被发出，没有早于6:00:31的元组可以到达。

三个窗口将被评估。通过采取最早事件时间戳(06:00:03)并基于滑动间隔(10s)计算

上限来计算第一个窗口结束在 `ts` (06:00:10)。

1. `5:59:50 - 06:00:10` 有元组 `e1, e2, e3`
2. `6:00:00 - 06:00:20` 有元组 `e1, e2, e3, e4`
3. `6:00:10 - 06:00:30` 有元组 `e4, e5`

`e6`未被评估，因为 `watermark` 时间戳`6:00:31`比元组 `ts``6:00:36`更旧。

在`9:00:01`和`9:00:02`之间，接收到的元组`e7(8:00:25), e8(8:00:26), e9(8:00:27), e10(8:00:39)`

在 `time t = 09:00:02`另一个 `watermark w2 = 08:00:34`被发出，没有元组比`8:00:34`更早到达。

三个窗口将被评估

1. `6:00:20 - 06:00:40` 有元组 `e5, e6` (从早期批次)
2. `6:00:30 - 06:00:50` 有元组 `e6` (从早期批次)
3. `8:00:10 - 08:00:30` 有元组 `e7, e8, e9`

`e10`不被评估，因为元组 `ts` `8:00:39`超出了 `watermark time` `8:00:34`.

窗口计算考虑时间间隔，并基于元组时间戳计算窗口。

## Guarantees

storm core的窗口功能目前提供一致性保证。执行(`TupleWindow inputWindow`)方法发出的值将自动锁定到 `inputWindow` 中的所有元组。预计下游 `bolts` 将确认接收的元组（即从窗口 `bolt` 发出的元组）以完成元组树。如果不是，元组将重播，并且重新评估窗口计算。

窗口中的元组会在过期后被自动确认，即当它们在`windowLength + slidingInterval`之后从窗口中滑落出来。请注意，配置`topology.message.timeout.secs`应该远远超过基于时间窗口的`windowLength + slidingInterval`；否则元组将超时并重播，并可能导致重复的评估。对于基于计数的窗口，应该调整配置，使得在超时间段内可以接收到`windowLength + slidingInterval`元组。

## 拓扑示例

示例拓扑`滑动窗口拓扑`显示了如何使用`apis`来计算滑动窗口总和和滚动窗口平均值。

# Joining Streams in Storm Core

Storm 支持通过 JoinBolt 来 join 多个 data streams 变成一个 stream. JoinBolt 是一个 Windowed bolt。JoinBolt 会等待配置的窗口时间来匹配被join 的streams的tuples。这有助于通过窗口边界生成streams.

JoinBolt 每个进来的 data streams 必须基于一个字段进行 Field Group。stream只能使用被 FieldsGrouped 的字段 join 其他stream。

## Performing Joins

考虑下面的 SQL join，设计四张表：

```
select userId, key4, key2, key3
from      table1
inner join table2 on table2.userId = table1.key1
inner join table3 on table3.key3    = table2.userId
left join  table4 on table4.key4    = table3.key3
```

相同的可以使用 JoinBolt，join 四个 spouts，生成想要的 tuples:

```
JoinBolt jbolt = new JoinBolt("spout1", "key1")                                // from      spout1
    .join      ("spout2", "userId",   "spout1")        // inner join spout2 on spout2
    .join      ("spout3", "key3",     "spout2")        // inner join spout3 on spout3
    .leftJoin ("spout4", "key4",     "spout3")        // left join  spout4 on spout4
    .select   ("userId, key4, key2, spout3:key3") // chose output fields
    .withTumblingWindow( new Duration(10, TimeUnit.MINUTES) ) ;
```

```
topoBuilder.setBolt("joiner", jbolt, 1)
    .fieldsGrouping("spout1", new Fields("key1") )
    .fieldsGrouping("spout2", new Fields("userId") )
    .fieldsGrouping("spout3", new Fields("key3") )
    .fieldsGrouping("spout4", new Fields("key4") );
```

bolt 构造器需要两个参数.第一个参数介绍了第一个 stream 来自于 spout1，并指定了通过 key1 来和其他 streams 连接.组件的名称必须根据直接连接 Join bolt 的 spout 或者 bolt 来设置.这里，来自于 spout1 的数据必须根据 key1 来 field group。同样的，调用 leftJoin() 和 join() 方法的时候，也会通过这个字段来 join.根据上面的例子，FieldGrouping 要求也适用于其他 spout 的 streams。第三个参数表示 streams 要和哪个 spout 的 streams 连接.

select() 方法用来指定 output fields。select 参数是逗号分隔的字段列表。单个字段可以通过 stream 名称作为前缀，来区别不同 streams 中相同的字段，像这样：.select("spout3:key3, spout4:key3").嵌套的 tuple 类型是支持的.例如，outer.inner.innermost

就是一个字段嵌套三层深度，`outer` 和 `inner` 是 `Map` 的类型.

`join` 参数中的字段不允许用 `stream` 名称作为前缀，但是支持嵌套字段.

上面调用 `withTumblingWindow()` 方法，将 `join` 窗口配置成 10 分钟的翻滚窗口. 由于 `JoinBolt` 是一个窗口 `spout`, 我们还可以使用 `withWindow` 方法将其配置为滑动窗口（参考下面的提示部分）.

## Stream names and Join order

- Streams name 在引用之前必须声明，在构造函数和 `join` 方法的第一个参数都需要 Streams name，`join` 方法的第三个参数会用到 Stream name. 像下面这样引用 stream name 是不允许的：

```
new JoinBolt( "spout1", "key1")
    .join      ( "spout2", "userId",   "spout3") //not allowed. spout3 not yet introduced
    .join      ( "spout3", "key3",     "spout1")
```

- 在内部，`join` 将按照用户所表示的顺序执行。

## Joining based on Stream names

为了简单起见，Storm topology（拓扑）经常使用 `default stream`。拓扑也可以使用命名的 stream 而不是 `default stream`。为了支持这种 topology（拓扑），可以通过构造函数的第一个参数将 `JoinBolt` 配置为使用 stream name 而不是源组件（`spout / bolt`）名称：

```
new JoinBolt(JoinBolt.Selector.STREAM,  "stream1", "key1")
    .join("stream2", "key2")
    ...
```

第一个参数 `JoinBolt.Selector.STREAM` 通知 bolt `stream1/2/3/4` 引用 named stream（而不是上游 `spouts/bolts` 的名称）。

以下示例从四个 `spouts` 连接两个命名流：

```
new JoinBolt(JoinBolt.Selector.STREAM,  "stream1", "key1")
    .join      ( "stream2", "userId",   "stream1" )
    .select   ("userId, key1, key2")
    .withTumblingWindow( new Duration(10, TimeUnit.MINUTES) ) ;

topoBuilder.setBolt("joiner", jbolt, 1)
    .fieldsGrouping("bolt1", "stream1", new Fields("key1"))
    .fieldsGrouping("bolt2", "stream1", new Fields("key1"))
    .fieldsGrouping("bolt3", "stream2", new Fields("userId"))
    .fieldsGrouping("bolt4", "stream1", new Fields("key1"));
```

在上述示例中，例如，`spout1`也可能发送其他 `stream`。但是连接 `bolt` 只是从不同的 `bolts` 订阅了 `stream1 & stream2`。来自 `bolt1`, `bolt2` 和 `bolt4` 的 `stream1` 被视为单个 `stream`，并且与 `bolt3` 相连接。

## Limitations:

1. 当前值支持 `INNER` 和 `LEFT join`。
  1. 不同于 `SQL`, 它允许通过不同的 `keys` 将相同的表和不同的表连接，这里必须在 `stream` 上使用相同的字段. `Fields Grouping` 保证 `tuples` 被正确连接到 `JoinBolt` 的实例. 因此，`FieldsGrouping` 字段必须与 `join` 字段相同，以获得正确的结果. 要在多个字段上执行 `join`, 可以将这些字段组着成一个字段，然后发送到 `Join Bolt`。

## Tips:

1. `Join` 可以是 `CPU` 和内存密集型. 当前窗口中积累的数据越大（与窗口长度成正比），`join` 所需要的时间就越长。滑动间隔很短（例如几秒钟）会触发频繁的连接. 因此，如果使用大的窗口长度或者小的滑动间隔，则性能可能受损.
2. 使用滑动窗口时，跨窗口重复 `join` 记录。这是因为使用滑动窗口时，`tuples` 在多个窗口中继续存在。
3. 如果启用了消息超时，请确保超时设置 (`topology.message.timeout.secs`) 足够大以舒适地适应窗口大小，以及其他 `spouts` 和 `bolts` 的附加处理。
4. 在最坏的情况下，连接一个具有 `M` 和 `N` 个元素的两个 `streams` 的窗口，可以产生每个输出元组的 `MxN` 元素，每个输出 `tuple` 从每个输入流锚定到一个 `tuple`。这可能意味着来自 `JoinBolt` 的大量输出元组和甚至更多的 `ACK` 用于下游 `spout` 发出。这可能会对消息传递系统造成重大压力，如果不小心，则会大大减缓 `topology` (拓扑) 结构。要管理消息传递子系统上的负载，建议：
  - 增加 `worker` 堆 (`topology.worker.max.heap.size.mb`)。
  - 如果您的 `topology` (拓扑) 不需要 `ACK`，则禁用 `ACKers` (`topology.acker.executors = 0`)。
  - 禁用事件记录器 (`topology.eventlogger.executors = 0`)。
  - 打开拓扑调试 (`topology.debug = false`)。
  - 将 `topology.max.spout` 设置为一个足够大的值，以容纳估计的全窗口值的 `tuple` 加上一

些更多的余量。这有助于减少在消息传递子系统遇到过载时发出过多元组的端口的可能性。当它的值设置为**null**时，可能会发生这种情况。

- 最后，将窗口大小保持在解决手头问题所需的最小值。

# Storm Distributed Cache API

---

## Storm Distributed Cache API

The distributed cache feature in storm is used to efficiently distribute files (or blobs, which is the equivalent terminology for a file in the distributed cache and is used interchangeably in this document) that are large and can change during the lifetime of a topology, such as geo-location data, dictionaries, etc. Typical use cases include phrase recognition, entity extraction, document classification, URL re-writing, location/address detection and so forth. Such files may be several KB to several GB in size. For small datasets that don't need dynamic updates, including them in the topology jar could be fine. But for large files, the startup times could become very large. In these cases, the distributed cache feature can provide fast topology startup, especially if the files were previously downloaded for the same submitter and are still in the cache. This is useful with frequent deployments, sometimes few times a day with updated jars, because the large cached files will remain available without changes. The large cached blobs that do not change frequently will remain available in the distributed cache.

At the starting time of a topology, the user specifies the set of files the topology needs. Once a topology is running, the user at any time can request for any file in the distributed cache to be updated with a newer version. The updating of blobs happens in an eventual consistency model. If the topology needs to know what version of a file it has access to, it is the responsibility of the user to find this information out. The files are stored in a cache with Least-Recently Used (LRU) eviction policy, where the supervisor decides which cached files are no longer needed and can delete them to free disk space. The blobs can be compressed, and the user can request the blobs to be uncompressed before it accesses them.

## Motivation for Distributed Cache

- Allows sharing blobs among topologies.
- Allows updating the blobs from the command line.

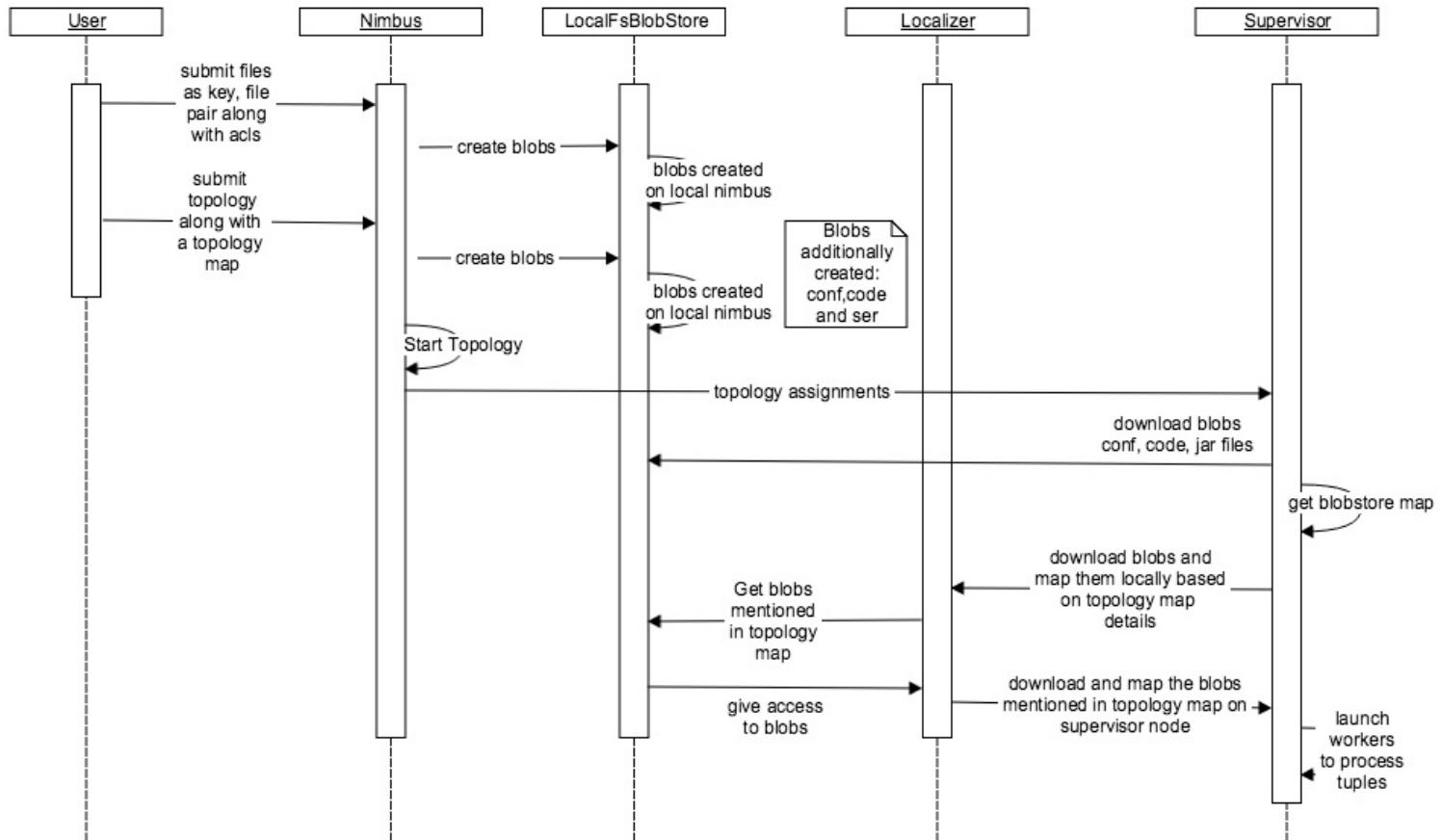
## Distributed Cache Implementations

The current BlobStore interface has the following two implementations \* LocalFsBlobStore \*

## HdfsBlobStore

Appendix A contains the interface for blobstore implementation.

## LocalFsBlobStore



Local file system implementation of Blobstore can be depicted in the above timeline diagram.

There are several stages from blob creation to blob download and corresponding execution of a topology. The main stages can be depicted as follows

### Blob Creation Command

Blobs in the blobstore can be created through command line using the following command.

```
storm blobstore create --file README.txt --acl o::rwa --replication-factor 4 key1
```

The above command creates a blob with a key name “key1” corresponding to the file README.txt. The access given to all users being read, write and admin with a replication factor of 4.

## Topology Submission and Blob Mapping

Users can submit their topology with the following command. The command includes the topology map configuration. The configuration holds two keys “key1” and “key2” with the key “key1” having a local file name mapping named “blob\_file” and it is not compressed.

```
storm jar /home/y/lib/storm-starter/current/storm-starter-jar-with-dependencies.jar  
org.apache.storm.starter.clj.word_count test_topo -c topology.blobstore.map='{"key1": {"localname": "
```

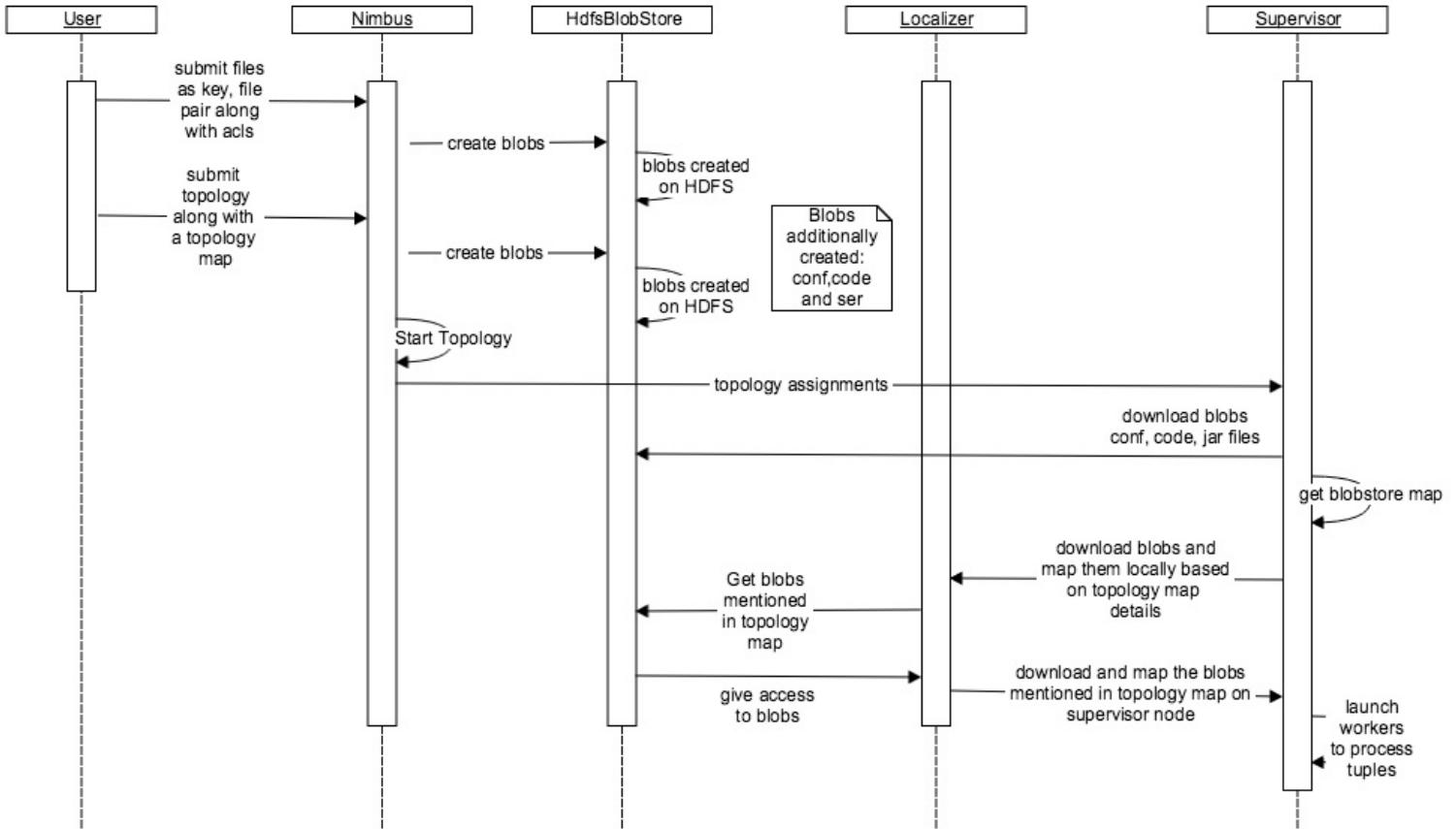
## Blob Creation Process

The creation of the blob takes place through the interface “ClientBlobStore”. Appendix B contains the “ClientBlobStore” interface. The concrete implementation of this interface is the “NimbusBlobStore”. In the case of local file system the client makes a call to the nimbus to create the blobs within the local file system. The nimbus uses the local file system implementation to create these blobs. When a user submits a topology, the jar, configuration and code files are uploaded as blobs with the help of blobstore. Also, all the other blobs specified by the topology are mapped to it with the help of topology.blobstore.map configuration.

## Blob Download by the Supervisor

Finally, the blobs corresponding to a topology are downloaded by the supervisor once it receives the assignments from the nimbus through the same “NimbusBlobStore” thrift client that uploaded the blobs. The supervisor downloads the code, jar and conf blobs by calling the “NimbusBlobStore” client directly while the blobs specified in the topology.blobstore.map are downloaded and mapped locally with the help of the Localizer. The Localizer talks to the “NimbusBlobStore” thrift client to download the blobs and adds the blob compression and local blob name mapping logic to suit the implementation of a topology. Once all the blobs have been downloaded the workers are launched to run the topologies.

## HdfsBlobStore



The HdfsBlobStore functionality has a similar implementation and blob creation and download procedure barring how the replication is handled in the two blobstore implementations. The replication in HDFS blobstore is obvious as HDFS is equipped to handle replication and it requires no state to be stored inside the zookeeper. On the other hand, the local file system blobstore requires the state to be stored on the zookeeper in order for it to work with nimbus HA. Nimbus HA allows the local filesystem to implement the replication feature seamlessly by storing the state in the zookeeper about the running topologies and syncing the blobs on various nimbuses. On the supervisor's end, the supervisor and localizer talks to HdfsBlobStore through "HdfsClientBlobStore" implementation.

## Additional Features and Documentation

```
storm jar /home/y/lib/storm-starter/current/storm-starter-jar-with-dependencies.jar org.apache.stor
-c topology.blobstore.map='{"key1":{"localname":"blob_file", "uncompress":false}, "key2":{}}'
```

### Compression

The blobstore allows the user to specify the "uncompress" configuration to true or false.

This configuration can be specified in the topology.blobstore.map mentioned in the above command. This allows the user to upload a compressed file like a tarball/zip. In local file system blobstore, the compressed blobs are stored on the nimbus node. The localizer code takes the responsibility to uncompress the blob and store it on the supervisor node. Symbolic links to the blobs on the supervisor node are created within the worker before the execution starts.

## Local File Name Mapping

Apart from compression the blobstore helps to give the blob a name that can be used by the workers. The localizer takes the responsibility of mapping the blob to a local name on the supervisor node.

## Additional Blobstore Implementation Details

Blobstore uses a hashing function to create the blobs based on the key. The blobs are generally stored inside the directory specified by the blobstore.dir configuration. By default, it is stored under “storm.local.dir/nimbus/blobs” for local file system and a similar path on hdfs file system.

Once a file is submitted, the blobstore reads the configs and creates a metadata for the blob with all the access control details. The metadata is generally used for authorization while accessing the blobs. The blob key and version contribute to the hash code and thereby the directory under “storm.local.dir/nimbus/blobs/data” where the data is placed. The blobs are generally placed in a positive number directory like 193,822 etc.

Once the topology is launched and the relevant blobs have been created, the supervisor downloads blobs related to the storm.conf, storm.ser and storm.code first and all the blobs uploaded by the command line separately using the localizer to uncompress and map them to a local name specified in the topology.blobstore.map configuration. The supervisor periodically updates blobs by checking for the change of version. This allows updating the blobs on the fly and thereby making it a very useful feature.

For a local file system, the distributed cache on the supervisor node is set to 10240 MB as a soft limit and the clean up code attempts to clean anything over the soft limit every 600 seconds based on LRU policy.

The HDFS blobstore implementation handles load better by removing the burden on the nimbus to store the blobs, which avoids it becoming a bottleneck. Moreover, it provides seamless replication of blobs. On the other hand, the local file system blobstore is not very efficient in replicating the blobs and is limited by the number of nimbuses. Moreover, the supervisor talks to the HDFS blobstore directly without the involvement of the nimbus and thereby reduces the load and dependency on nimbus.

## Highly Available Nimbus

### Problem Statement:

Currently the storm master aka nimbus, is a process that runs on a single machine under supervision. In most cases, the nimbus failure is transient and it is restarted by the process that does supervision. However sometimes when disks fail and networks partitions occur, nimbus goes down. Under these circumstances, the topologies run normally but no new topologies can be submitted, no existing topologies can be killed/deactivated/activated and if a supervisor node fails then the reassessments are not performed resulting in performance degradation or topology failures. With this project we intend, to resolve this problem by running nimbus in a primary backup mode to guarantee that even if a nimbus server fails one of the backups will take over.

### Requirements for Highly Available Nimbus:

- Increase overall availability of nimbus.
- Allow nimbus hosts to leave and join the cluster at will any time. A newly joined host should auto catch up and join the list of potential leaders automatically.
- No topology resubmissions required in case of nimbus fail overs.
- No active topology should ever be lost.

### Leader Election:

The nimbus server will use the following interface:

```
public interface ILeaderElector {  
    /**  
     * queue up for leadership lock. The call returns immediately and the caller  
     * must check isLeader() to perform any leadership action.  
     */  
    void addToLeaderLockQueue();
```

```

/**
 * Removes the caller from the leader lock queue. If the caller is leader
 * also releases the lock.
 */
void removeFromLeaderLockQueue();

/**
 *
 * @return true if the caller currently has the leader lock.
 */
boolean isLeader();

/**
 *
 * @return the current leader's address , throws exception if noone has has      lock.
 */
InetSocketAddress getLeaderAddress();

/**
 *
 * @return list of current nimbus addresses, includes leader.
 */
List<InetSocketAddress> getAllNimbusAddresses();
}

```

Once a nimbus comes up it calls addToLeaderLockQueue() function. The leader election code selects a leader from the queue. If the topology code, jar or config blobs are missing, it would download the blobs from any other nimbus which is up and running.

The first implementation will be Zookeeper based. If the zookeeper connection is lost/reset resulting in loss of lock or the spot in queue the implementation will take care of updating the state such that isLeader() will reflect the current status. The leader like actions must finish in less than minimumOf(connectionTimeout, SessionTimeout) to ensure the lock was held by nimbus for the entire duration of the action (Not sure if we want to just state this expectation and ensure that zk configurations are set high enough which will result in higher failover time or we actually want to create some sort of rollback mechanism for all actions, the second option needs a lot of code). If a nimbus that is not leader receives a request that only a leader can perform, it will throw a RunTimeException.

## Nimbus state store:

To achieve fail over from primary to backup servers nimbus state/data needs to be replicated across all nimbus hosts or needs to be stored in a distributed storage.

Replicating the data correctly involves state management, consistency checks and it is hard to test for correctness. However many storm users do not want to take extra dependency on another replicated storage system like HDFS and still need high availability. The blobstore implementation along with the state storage helps to overcome the failover scenarios in case a leader nimbus goes down.

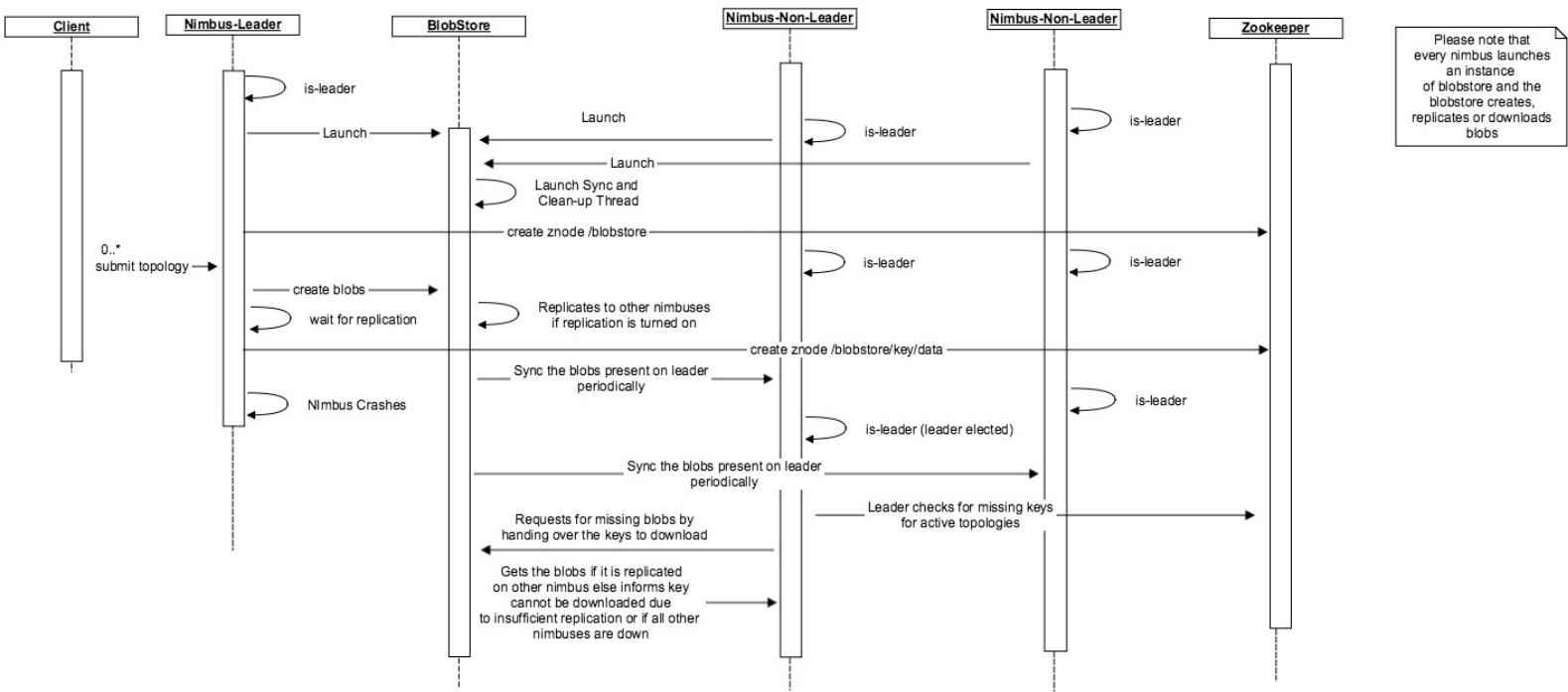
To support replication we will allow the user to define a code replication factor which would reflect number of nimbus hosts to which the code must be replicated before starting the topology. With replication comes the issue of consistency. The topology is launched once the code, jar and conf blob files are replicated based on the "topology.min.replication" config. Maintaining state for failover scenarios is important for local file system. The current implementation makes sure one of the available nimbus is elected as a leader in the case of a failure. If the topology specific blobs are missing, the leader nimbus tries to download them as and when they are needed. With this current architecture, we do not have to download all the blobs required for a topology for a nimbus to accept leadership. This helps us in case the blobs are very large and avoid causing any inadvertent delays in electing a leader.

The state for every blob is relevant for the local blobstore implementation. For HDFS blobstore the replication is taken care by the HDFS. For handling the fail over scenarios for a local blobstore we need to store the state of the leader and non-leader nimbuses within the zookeeper.

The state is stored under /storm/blobstore/key/nimbusHostPort:SequenceNumber for the blobstore to work to make nimbus highly available. This state is used in the local file system blobstore to support replication. The HDFS blobstore does not have to store the state inside the zookeeper.

- **NimbusHostPort:** This piece of information generally contains the parsed string holding the hostname and port of the nimbus. It uses the same class “NimbusHostPortInfo” used earlier by the code-distributor interface to store the state and parse the data.
- **SequenceNumber:** This is the blob sequence number information. The SequenceNumber information is implemented by a KeySequenceNumber class. The sequence numbers are generated for every key. For every update, the sequence numbers are assigned

based on a global sequence number stored under /storm/blobstoremaxsequencenumber/key. For more details about how the numbers are generated you can look at the java docs for KeySequenceNumber.



The sequence diagram proposes how the blobstore works and the state storage inside the zookeeper makes the nimbus highly available. Currently, the thread to sync the blobs on a non-leader is within the nimbus. In the future, it will be nice to move the thread around to the blobstore to make the blobstore coordinate the state change and blob download as per the sequence diagram.

## Thrift and Rest API

In order to avoid workers/supervisors/ui talking to zookeeper for getting master nimbus address we are going to modify the `getClusterInfo` API so it can also return nimbus information. `getClusterInfo` currently returns `ClusterSummary` instance which has a list of `supervisorSummary` and a list of `topologySummary` instances. We will add a list of `NimbusSummary` to the `ClusterSummary`. See the structures below:

```

struct ClusterSummary {
    1: required list<SupervisorSummary> supervisors;
    3: required list<TopologySummary> topologies;
    4: required list<NimbusSummary> nimbuses;
}

struct NimbusSummary {
    1: required string host;
}

```

```
2: required i32 port;
3: required i32 uptime_secs;
4: required bool isLeader;
5: required string version;
}
```

This will be used by StormSubmitter, Nimbus clients, supervisors and ui to discover the current leaders and participating nimbus hosts. Any nimbus host will be able to respond to these requests. The nimbus hosts can read this information once from zookeeper and cache it and keep updating the cache when the watchers are fired to indicate any changes, which should be rare in general case.

Note: All nimbus hosts have watchers on zookeeper to be notified immediately as soon as a new blobs is available for download, the callback may or may not download the code. Therefore, a background thread is triggered to download the respective blobs to run the topologies. The replication is achieved when the blobs are downloaded onto non-leader nimbuses. So you should expect your topology submission time to be somewhere between 0 to  $(2 * \text{nimbus.code.sync.freq.secs})$  for any `nimbus.min.replication.count > 1`.

## Configuration

`blobstore.dir`: The directory where all blobs are stored. For local file system it represents the directory and for HDFS file system it represents the hdfs file system path.

`supervisor.blobstore.class`: This configuration is meant to set the client for the supervisor in order. For a local file system blobstore it is set to "org.apache.storm.blobstore.NimbusBlobStore" and for HDFS it is set to "org.apache.storm.blobstore.HdfsClientBlobStore".

`supervisor.blobstore.download.thread.count`: This configuration spawns multiple threads for downloading the blobs concurrently. The default is set to 5.

`supervisor.blobstore.download.max_retries`: This configuration is set to allow the supervisor to retry failed downloads. By default it is set to 3.

`supervisor.localizer.cache.target.size.mb`: The jvm opts provided to workers launched by this supervisor are replaced with an identifier for this worker. Also, "%WORKER-ID%", "%STORM-ID%" and "%WORKER-PORT%" are appropriate runtime values for this worker. The distributed cache target size in MB. This is a soft limit of the distributed cache contents. It is set to 10240 MB.

`supervisor.localizer.cleanup.interval.ms`: The distributed cache cleanup interval. Controls how often the supervisor cleanup anything over the cache target size. By default it is set to 600000 milliseconds.

`nimbus.blobstore.class`: Sets the blobstore implementation nimbus uses. It is set to "org.apache.storm.blobstore.NimbusBlobStore".

`nimbus.blobstore.expiration.secs`: During operations with the blobstore, via master, how long a connection considers it dead and drops the session and any associated connections. The default is set to 600.

`storm.blobstore.inputstream.buffer.size.bytes`: The buffer size it uses for blobstore upload. It is set to 1024000000 bytes.

`client.blobstore.class`: The blobstore implementation the storm client uses. The current implementation config "org.apache.storm.blobstore.NimbusBlobStore".

`blobstore.replication.factor`: It sets the replication for each blob within the blobstore. The "topo ensures the minimum replication the topology specific blobs are set before launching the topology. "topology.min.replication.count <= blobstore.replication". The default is set to 3.

`topology.min.replication.count` : Minimum number of nimbus hosts where the code must be replicated before can mark the topology as active and create assignments. Default is 1.

`topology.max.replication.wait.time.sec`: Maximum wait time for the nimbus host replication to achieve Once this time is elapsed nimbus will go ahead and perform topology activation tasks even if required. The default is 60 seconds, a value of -1 indicates to wait for ever.

\* `nimbus.code.sync.freq.secs`: Frequency at which the background thread on nimbus which syncs code from

## Using the Distributed Cache API, Command Line Interface (CLI)

### Creating blobs

To use the distributed cache feature, the user first has to "introduce" files that need to be cached and bind them to key strings. To achieve this, the user uses the "blobstore create" command of the storm executable, as follows:

```
storm blobstore create [-f|--file FILE] [-a|--acl ACL1,ACL2,...] [--replication-factor NUMBER] [key]
```

The contents come from a FILE, if provided by -f or --file option, otherwise from STDIN. The ACLs, which can also be a comma separated list of many ACLs, is of the following format:

```
> [u|o]:[username]:[r-|w-|a-|_]
```

where:

- u = user
- o = other
- username = user for this particular ACL
- r = read access
- w = write access
- a = admin access
- \_ = ignored

The replication factor can be set to a value greater than 1 using --replication-factor.

Note: The replication right now is configurable for a hdfs blobstore but for a local blobstore the replication always stays at 1. For a hdfs blobstore the default replication is set to 3.

### Example:

```
storm blobstore create --file README.txt --acl o::rwa --replication-factor 4 key1
```

In the above example, the *README.txt* file is added to the distributed cache. It can be accessed using the key string "key1" for any topology that needs it. The file is set to have read/write/admin access for others, a.k.a world everything and the replication is set to 4.

### Example:

```
storm blobstore create mytopo:data.tgz -f data.tgz -a u:alice:rwa,u:bob:rw,o::r
```

The above example creates a mytopo:data.tgz key using the data stored in data.tgz. User alice would have full access, bob would have read/write access and everyone else would have read access.

## Making dist. cache files accessible to topologies

Once a blob is created, we can use it for topologies. This is generally achieved by including the key string among the configurations of a topology, with the following format. A shortcut is to add the configuration item on the command line when starting a topology by using the -c command:

```
-c topology.blobstore.map='{"[KEY]":{"localname":"[VALUE]", "uncompress": "[true|false]"}'}
```

Note: Please take care of the quotes.

The cache file would then be accessible to the topology as a local file with the name [VALUE].

The localname parameter is optional, if omitted the local cached file will have the same name as [KEY].

The uncompress parameter is optional, if omitted the local cached file will not be uncompressed. Note that the key string needs to have the appropriate file-name-like format

and extension, so it can be uncompressed correctly.

## Example:

```
storm jar /home/y/lib/storm-starter/current/storm-starter-jar-with-dependencies.jar org.apache.stor
```

Note: Please take care of the quotes.

In the above example, we start the *word\_count* topology (stored in the *storm-starter-jar-with-dependencies.jar* file), and ask it to have access to the cached file stored with key string = *key1*. This file would then be accessible to the topology as a local file called *blob\_file*, and the supervisor will not try to uncompress the file. Note that in our example, the file's content originally came from *README.txt*. We also ask for the file stored with the key string = *key2* to be accessible to the topology. Since both the optional parameters are omitted, this file will get the local name = *key2*, and will not be uncompressed.

## Updating a cached file

It is possible for the cached files to be updated while topologies are running. The update happens in an eventual consistency model, where the supervisors poll Nimbus every 30 seconds, and update their local copies. In the current version, it is the user's responsibility to check whether a new file is available.

To update a cached file, use the following command. Contents come from a FILE or STDIN. Write access is required to be able to update a cached file.

```
storm blobstore update [-f|--file NEW_FILE] [KEYSTRING]
```

## Example:

```
storm blobstore update -f updates.txt key1
```

In the above example, the topologies will be presented with the contents of the file *updates.txt* instead of *README.txt* (from the previous example), even though their access by the topology is still through a file called *blob\_file*.

## Removing a cached file

To remove a file from the distributed cache, use the following command. Removing a file requires write access.

```
storm blobstore delete [KEYSTRING]
```

## **Listing Blobs currently in the distributed cache blobstore**

```
storm blobstore list [KEY...]
```

lists blobs currently in the blobstore

## **Reading the contents of a blob**

```
storm blobstore cat [-f|--file FILE] KEY
```

read a blob and then either write it to a file, or STDOUT. Reading a blob requires read access.

## **Setting the access control for a blob**

```
set-acl [-s ACL] KEY
```

ACL is in the form [uo]:[username]:[r-][w-][a-] can be comma separated list (requires admin access).

## **Update the replication factor for a blob**

```
storm blobstore replication --update --replication-factor 5 key1
```

## **Read the replication factor of a blob**

```
storm blobstore replication --read key1
```

## **Command line help**

```
storm help blobstore
```

## **Using the Distributed Cache API from Java**

We start by getting a ClientBlobStore object by calling this function:

```
Config theconf = new Config();
```

```
theconf.putAll(Utils.readStormConfig());
ClientBlobStore clientBlobStore = Utils.getClientBlobStore(theconf);
```

The required Utils package can be imported by:

```
import org.apache.storm.utils.Utils;
```

ClientBlobStore and other blob-related classes can be imported by:

```
import org.apache.storm.blobstore.ClientBlobStore;
import org.apache.storm.blobstore.AtomicOutputStream;
import org.apache.storm.blobstore.InputStreamWithMeta;
import org.apache.storm.blobstore.BlobStoreAclHandler;
import org.apache.storm.generated.*;
```

## Creating ACLs to be used for blobs

```
String stringBlobACL = "u:username:rwa";
AccessControl blobACL = BlobStoreAclHandler.parseAccessControl(stringBlobACL);
List<AccessControl> acls = new LinkedList<AccessControl>();
acls.add(blobACL); // more ACLs can be added here
SettableBlobMeta settableBlobMeta = new SettableBlobMeta(acls);
settableBlobMeta.set_replication_factor(4); // Here we can set the replication factor
```

The settableBlobMeta object is what we need to create a blob in the next step.

## Creating a blob

```
AtomicOutputStream blobStream = clientBlobStore.createBlob("some_key", settableBlobMeta);
blobStream.write("Some String or input data".getBytes());
blobStream.close();
```

Note that the settableBlobMeta object here comes from the last step, creating ACLs. It is recommended that for very large files, the user writes the bytes in smaller chunks (for example 64 KB, up to 1 MB chunks).

## Updating a blob

Similar to creating a blob, but we get the AtomicOutputStream in a different way:

```
String blobKey = "some_key";
AtomicOutputStream blobStream = clientBlobStore.updateBlob(blobKey);
```

Pass a byte stream to the returned AtomicOutputStream as before.

## Updating the ACLs of a blob

```

String blobKey = "some_key";
AccessControl updateAcl = BlobStoreAclHandler.parseAccessControl("u:USER:--a");
List<AccessControl> updateAccls = new LinkedList<AccessControl>();
updateAccls.add(updateAcl);
SettableBlobMeta modifiedSettableBlobMeta = new SettableBlobMeta(updateAccls);
clientBlobStore.setBlobMeta(blobKey, modifiedSettableBlobMeta);

//Now set write only
updateAcl = BlobStoreAclHandler.parseAccessControl("u:USER:-w-");
updateAccls = new LinkedList<AccessControl>();
updateAccls.add(updateAcl);
modifiedSettableBlobMeta = new SettableBlobMeta(updateAccls);
clientBlobStore.setBlobMeta(blobKey, modifiedSettableBlobMeta);

```

## Updating and Reading the replication of a blob

```

String blobKey = "some_key";
BlobReplication replication = clientBlobStore.updateBlobReplication(blobKey, 5);
int replication_factor = replication.get_replication();

```

Note: The replication factor gets updated and reflected only for hdfs blobstore

## Reading a blob

```

String blobKey = "some_key";
InputStreamWithMeta blobInputStream = clientBlobStore.getBlob(blobKey);
BufferedReader r = new BufferedReader(new InputStreamReader(blobInputStream));
String blobContents = r.readLine();

```

## Deleting a blob

```

String blobKey = "some_key";
clientBlobStore.deleteBlob(blobKey);

```

## Getting a list of blob keys already in the blobstore

```

Iterator <String> stringIterator = clientBlobStore.listKeys();

```

## Appendix A

```

public abstract void prepare(Map conf, String baseDir);

public abstract AtomicOutputStream createBlob(String key, SettableBlobMeta meta, Subject who) throw
public abstract AtomicOutputStream updateBlob(String key, Subject who) throws AuthorizationException
public abstract ReadableBlobMeta getBlobMeta(String key, Subject who) throws AuthorizationException
public abstract void setBlobMeta(String key, SettableBlobMeta meta, Subject who) throws Authorizati

```

```
public abstract void deleteBlob(String key, Subject who) throws AuthorizationException, KeyNotFoundException  
public abstract InputStreamWithMeta getBlob(String key, Subject who) throws AuthorizationException,  
public abstract Iterator<String> listKeys(Subject who);  
public abstract BlobReplication getBlobReplication(String key, Subject who) throws Exception;  
public abstract BlobReplication updateBlobReplication(String key, int replication, Subject who) throws Exception;
```

## Appendix B

```
public abstract void prepare(Map conf);  
  
protected abstract AtomicOutputStream createBlobToExtend(String key, SettableBlobMeta meta) throws AuthorizationException, KeyNotFoundException;  
public abstract AtomicOutputStream updateBlob(String key) throws AuthorizationException, KeyNotFoundException;  
public abstract ReadableBlobMeta getBlobMeta(String key) throws AuthorizationException, KeyNotFoundException;  
protected abstract void setBlobMetaToExtend(String key, SettableBlobMeta meta) throws AuthorizationException, KeyNotFoundException;  
public abstract void deleteBlob(String key) throws AuthorizationException, KeyNotFoundException;  
public abstract InputStreamWithMeta getBlob(String key) throws AuthorizationException, KeyNotFoundException;  
public abstract Iterator<String> listKeys();  
public abstract void watchBlob(String key, IBlobWatcher watcher) throws AuthorizationException;  
public abstract void stopWatchingBlob(String key) throws AuthorizationException;  
public abstract BlobReplication getBlobReplication(String key) throws AuthorizationException, KeyNotFoundException;  
public abstract BlobReplication updateBlobReplication(String key, int replication) throws AuthorizationException, KeyNotFoundException;
```

## Appendix C

```
service Nimbus {  
    ...  
    string beginCreateBlob(1: string key, 2: SettableBlobMeta meta) throws (1: AuthorizationException aze, 2: KeyNotFoundException aze);  
    string beginUpdateBlob(1: string key) throws (1: AuthorizationException aze, 2: KeyNotFoundException aze);  
    void uploadBlobChunk(1: string session, 2: binary chunk) throws (1: AuthorizationException aze);  
    void finishBlobUpload(1: string session) throws (1: AuthorizationException aze);  
    void cancelBlobUpload(1: string session) throws (1: AuthorizationException aze);  
    ReadableBlobMeta getBlobMeta(1: string key) throws (1: AuthorizationException aze, 2: KeyNotFoundException aze);  
}
```

```
void setBlobMeta(1: string key, 2: SettableBlobMeta meta) throws (1: AuthorizationException aze, 2:  
BeginDownloadResult beginBlobDownload(1: string key) throws (1: AuthorizationException aze, 2: KeyN  
binary downloadBlobChunk(1: string session) throws (1: AuthorizationException aze);  
void deleteBlob(1: string key) throws (1: AuthorizationException aze, 2: KeyNotFoundException knf);  
ListBlobsResult listBlobs(1: string session);  
BlobReplication getBlobReplication(1: string key) throws (1: AuthorizationException aze, 2: KeyNotF  
BlobReplication updateBlobReplication(1: string key, 2: i32 replication) throws (1: AuthorizationEx  
...  
}  
  
struct BlobReplication {  
1: required i32 replication;  
}  
  
exception AuthorizationException {  
1: required string msg;  
}  
  
exception KeyNotFoundException {  
1: required string msg;  
}  
  
exception KeyAlreadyExistsException {  
1: required string msg;  
}  
  
enum AccessControlType {  
OTHER = 1,  
USER = 2  
//eventually ,GROUP=3  
}  
  
struct AccessControl {  
1: required AccessControlType type;  
2: optional string name; //Name of user or group in ACL  
3: required i32 access; //bitmasks READ=0x1, WRITE=0x2, ADMIN=0x4  
}  
  
struct SettableBlobMeta {  
1: required list<AccessControl> acl;  
2: optional i32 replication_factor  
}  
  
struct ReadableBlobMeta {  
1: required SettableBlobMeta settable;  
//This is some indication of a version of a BLOB. The only guarantee is  
// if the data changed in the blob the version will be different.  
2: required i64 version;  
}  
  
struct ListBlobsResult {  
1: required list<string> keys;
```

```
2: required string session;
}

struct BeginDownloadResult {
//Same version as in ReadableBlobMeta
1: required i64 version;
2: required string session;
3: optional i64 data_size;
}
```

# Storm 调试

# 动态日志级别设置

我们已经添加了使用Storm UI 和Storm CLI为正在运行的拓扑设置日志级别的功能。

日志级别设置的应用方式与log4j所期望的相同，因为我们正在做的是告诉log4j设置您提供的记录器的级别。如果您设置父记录器的日志级别，则子级记录器将开始用该级别(除非该子级别的限制级别更高)。可以选择提供超时（除了DEBUG模式，在UI中需要它了），如果工作人员应自动重置日志级别。

这种恢复操作是使用轮询机制触发的（每隔30秒，但这是可配置的），所以你应该期望你的超时值是你提供的值加上0和设置值之间的任何值。

## Using the Storm UI

### 使用 Storm UI

为了设置一个级别，请单击运行的拓扑，然后单击"拓扑操作"部分中的"更改日志级别"。

#### Topology actions

Activate   Deactivate   Rebalance   Kill   Change Log Level

#### Change Log Level

Modify the logger levels for topology. Note that applying a setting restarts the timer in the workers. To configure the root logger, use the name ROOT.

Logger	Level	Timeout (sec)	Expires at	Actions
com.myapp	DEBUG	30	4/7/2015 1:43:49 PM	<button>Apply</button> <button>Clear</button>
com.your.organization.Lo	Pick Level	30		<button>Add</button>

然后，提供记录器的名称，选择您期望的级别（例如WARN）和超时（以秒为单位）（如果不需则为0），然后点击"添加"。

#### Topology actions

Activate   Deactivate   Rebalance   Kill   Change Log Level

#### Change Log Level

Modify the logger levels for topology. Note that applying a setting restarts the timer in the workers. To configure the root logger, use the name ROOT.

Logger	Level	Timeout (sec)	Expires at	Actions
com.your.organization.Lo	Pick Level	30		<button>Add</button>

要清理日志级别，请单击"清除"按钮。这会将日志级别恢复添加设置之前的级别。日志级别线也将从UI消失。

虽然有延时重置日志级别，但首先设置日志级别是即时消息（或者消息可以通过nimbus和zookeeper从UI/CLI传送到工作人员）。

## 使用 **CLI**

使用CLI发出命令：

```
./bin/storm set_log_level [topology name] -l [logger name]=[LEVEL]:[TIMEOUT]
```

例如：

```
./bin/storm set_log_level my_topology -l ROOT=DEBUG:30
```

将ROOT记录器设置为DEBUG 30秒。

```
./bin/storm set_log_level my_topology -r ROOT
```

清除ROOT记录器动态日志级别，将其重置为原始值。

# Storm Logs

日志在storm中对于跟踪状态、操作、错误信息和调试信息至关重要 对于所有的守护进程(e.g.,nimbus, supervisor, logviewer, drpc, ui, pacemaker)和拓扑作业人员也是一样重要。

## 日志的位置

所有的守护进程都会在\${storm.log.dir}这个目录下面，管理员可以在系统属性或者在集群中配置，默认， \${storm.log.dir} 指向的是\${storm.home}/logs目录.

所有的工作日志的位置在worker-artifacts目录下面以分级的方式存在，例如， \${workers-artifacts}/\${topologyId}/\${port}/workder.log. 用户可以通过配置参数"storm.workers.artifacts.dir"来设置worder-artifacts目录的位置，其中， worker-artifacts目录的默认位置是\${storm.log.dir}/logs/workers-artifacts.

## 使用**storm UI** 进行日志查看/下载和日志搜索

授权用户允许守护进程和工作日志通过**Storm UI** 进行查看和下载

为了改善**Storm**的调试，我们提供了**log Search**的功能. **Log Search** 支持在某些日志文件或是在所有的拓扑日志文件中搜索： 字符串搜索日志文件： 在工作日志页面中，用户可以在某个工作日志中搜索某些字符串，比如：“Exception”。这种搜索方式通常会发生在正常文本日志或滚动的zip日志文件中。在结果中将会显示出偏移和匹配的行数。

Most Visited

Yahoo!

Yahoo! Backyard

Yahoo! IT Support

zhuol (Zhuo Liu)

Search wc6-1-1446571009/6700/worker.log: INFO

Search

Next

## File offset Match

38

2015-11-03 11:17:00.164 b.s.d.worker [INFO] Launching worker for wc6-1-1446571009 on 86b9c113-339e-4b8f

7157

simple-acl.users.commands" [], "drpc.request.timeout.secs" 600}

2015-11-03 11:17:00.367 o.a.s.s.o.a.c.f.i.CuratorFrameworkImpl [INFO] Starting

2015-11-03 11:17:00.376 o.a.s.s.o.a.z.ZooKeeper [INFO] Client environment:zookeeper.version=3.4.6-15699

7221

2015-11-03 11:17:00.367 o.a.s.s.o.a.c.f.i.CuratorFrameworkImpl [INFO] Starting

2015-11-03 11:17:00.376 o.a.s.s.o.a.z.ZooKeeper [INFO] Client environment:zookeeper.version=3.4.6-15699

2015-11-03 11:17:00.376 o.a.s.s.o.a.z.Zookee

7358

ent environment:zookeeper.version=3.4.6-1569965, built on 02/20/2014 09:09 GMT

2015-11-03 11:17:00.376 o.a.s.s.o.a.z.ZooKeeper [INFO] Client environment:host.name=shorelane-lm.champ.

2015-11-03 11:17:00.376 o.a.s.s.o.a.z.ZooKeeper [INFO] Client e

7476

ooKeeper [INFO] Client environment:host.name=shorelane-lm.champ.corp.yahoo.com

2015-11-03 11:17:00.376 o.a.s.s.o.a.z.ZooKeeper [INFO] Client environment:java.version=1.8.0\_60

2015-11-03 11:17:00.377 o.a.s.s.o.a.z.ZooKeeper [INFO] Client environment:java.vendor

在拓扑中搜索：用户同时也可以通过单击UI页面的右上角的放大镜图标来某个拓扑的字符串。这意味着UI将尝试着以分布式的方式在所有主节点上搜索，以便在此拓扑的所有日志中查找匹配的字符串。通过检查/取消选中"搜索归档日志": box，可以对普通文本日志文件或滚动的zip日志文件进行搜索。然后将匹配的结果显示在具有url链接的UI页面上，将用户指向每个主节点上的某些日志。这个强大的功能非常有助于用户找到运行此拓扑上的某些有问题的主节点。

Topology Id:

wc6-1-1446571009

Search:

INFO

Search archived logs:



Search

host:port	Match
<a href="#">shorelane-lm.champ.corp.yahoo.com:6700</a>	<pre>2015-11-03 11:17:00.164 b.s.d.worker [INFO] Launching worker for wc6-1-1446571009</pre>
<a href="#">shorelane-lm.champ.corp.yahoo.com:6700</a>	<pre>simple-acl.users.commands" [], "drpc.request.timeout.secs" 600} 2015-11-03 11:17:00.367 o.a.s.s.o.a.c.f.i.CuratorFrameworkImpl [INFO] Starting 2015-11-03 11:17:00.376 o.a.s.s.o.a.z.ZooKeeper [INFO] Client environment:zookeeper</pre>
<a href="#">shorelane-lm.champ.corp.yahoo.com:6701</a>	<pre>2015-11-03 11:17:00.226 b.s.d.worker [INFO] Launching worker for wc6-1-1446571009</pre>
<a href="#">shorelane-lm.champ.corp.yahoo.com:6701</a>	<pre>simple-acl.users.commands" [], "drpc.request.timeout.secs" 600} 2015-11-03 11:17:00.411 o.a.s.s.o.a.c.f.i.CuratorFrameworkImpl [INFO] Starting 2015-11-03 11:17:00.419 o.a.s.s.o.a.z.ZooKeeper [INFO] Client environment:zookeeper</pre>
<a href="#">shorelane-lm.champ.corp.yahoo.com:6702</a>	<pre>2015-11-03 11:17:00.237 b.s.d.worker [INFO] Launching worker for wc6-1-1446571009</pre>

# 动态员工分析

在多组户模式下， storm通过集群启动长时间运行的JVM， 而无需sudo访问用户。 Java堆， jstacks和Java分析这些JVM的自服务将提高用户在主动监听时分析和调试问题的能力。

storm 动态分析器可以让您动态地对库存群集上运行的JVM进行head-dump,jprofiler或jstack. 它让用户从浏览器下载这些转存，并使用您最喜爱的工具进行分析。 UI组件页面为组件和操作按钮提供列表工作人员。 Logviewer可以让您下载这些日志生成的转储， 有关详细信息，请参阅截图。

## 使用 Storm UI

为了请求堆转储， jstack,启动/停止/转储jprofile或重新启动一个工作者， 点击运行的拓扑， 然后点击特定的组件， 然后您可以通过选中任何工作人员的执行者的框来选择工人 执行程序表， 然后在“分析和调试”部分中单击“开始”， “堆”， “堆栈”或“人工重启”。

### Executors (All time)

Show		20	entries	Search:						
Id	Uptime	Host		Port	Actions	Emitted	Transferred	Complete latency (ms)	Acked	Failed
[27-27]	8m 21s	host0		6702	<input checked="" type="checkbox"/> files	4760	4760	0.000	0	0
[28-28]	2h 47m 25s	host0		6700	<input type="checkbox"/> files	96160	96160	0.000	0	0
[29-29]	8m 27s	host0		6701	<input type="checkbox"/> files	4740	4740	0.000	0	0
[30-30]	8m 21s	host0		6702	<input checked="" type="checkbox"/> files	4780	4780	0.000	0	0

在 Executors表中， 单击任何执行程序旁边的“操作”列中的复选框，并且自动选择属于同一个工作的任何其他执行程序。操作完成后， 创建的任何文件输出文件将在“操作”列中的链接处可用。

## Profiling and Debugging

Use the following controls to profile and debug the components on this page.

Status / Timeout (Minutes)	Actions
10	<button>Start</button> <button>JStack</button> <button>Restart Worker</button> <button>Heap</button>

对于启动jprofile, 提供以分钟为单位的超时（如果不需要则为10）。然后点击"开始"。

# Profiling and Debugging

Use the following controls to profile and debug the components on this page.

Status / Timeout (Minutes)	Actions			
10	Start	JStack	Restart Worker	Heap
host0:6702 Active until 22:58:19 GMT-0600 (CST)	Stop	Dump Profile	JStack	Restart Worker
	Heap	<a href="#">My Dump Files</a>		

要停止jprofile日志记录，单击“停止”按钮。这将转储jprofile统计信息并停止分析。刷新该行的页面从UI消失。

单击“我的转储文件”，以转到用于特定于工作的转储文件列表的日志查看器UI。

[survivedlived.corp.ir2.yahoo.com:8000/dumps/wordcount-1-1446484724/survivedlived.corp.ir2.yahoo.com%3A6701](#)

- [jstack-6168-20151102-172741.txt](#)
- [recording-6168-1-20151102-172741.jfr](#)
- [recording-6168-20151102-172741.bin](#)

## 配置

可以将“worker.profiler.command”配置为指向特定的可插拔分析器，`heapdump`命令。如果插件不可用或jdk不支持JProfile航班录制，“worker.profiler.enabled”可以补禁用，以便工作JVM选项不会有“worker.profiler.childopts”。要使用不同的profiler插件，您可以更改这些配置。

# 拓扑事件检查器

## 简介

拓扑事件检查器提供了在storm拓扑的不同阶段时查看元组的功能。这可以用于在拓扑运行时检查在拓扑管线中的a spout(喷口)或a bolt(螺栓)处发射的元组，而不用停止或重新部署拓扑。从the spouts(喷口)到the bolts(螺栓)元组的正常流动是不受找开事件记录的影响。

## 启用事件日志记录

注意：首先事件日志记录需要将storm的"topology.eventlogger.executors"参数设置成非零的值。详情请查询 [Configuration](#) 章节内容。

可以通过在拓扑视图中的拓扑操作下点击“调试”按钮来记录事件。这会记录来自所有spouts（喷口）和bolts（螺栓）的元组以指定的采样百分比在拓扑中。

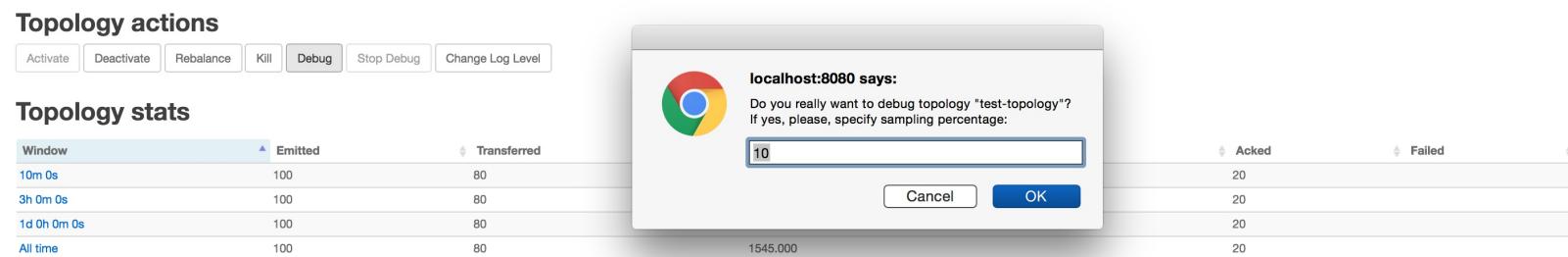


Figure 1: Enable event logging at topology level.

您还可以通过转到相应的组件页面来启用特定(spout)喷口或(bolt)螺栓级别的事件记录和单击组件操作下的“调试”。

## Storm UI

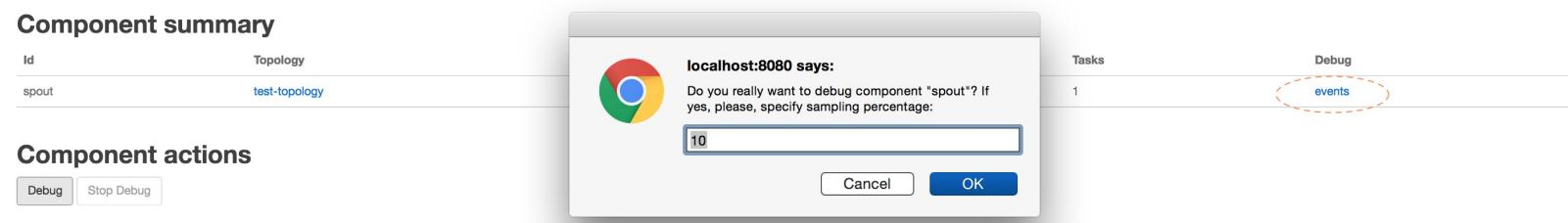


Figure 2: Enable event logging at component level.

## 查看事件日志

Storm "logviewer" 应该运行查看已记录的元组。如果没有运行，则可以从Storm安装目录运行“bin/storm logviewer”命令启动日志查看器。要查看元组，请从Storm UI中访问特定的spout(喷口)或bolt(螺栓)组件页面，然后单击组件摘要下的“事件”链接(如上图2所示)。

这将打开一个如下所示的视图，您可以在不同的页面之间导航并查看自己已记录的元组。

### test-topology-1-1459147817/6702/events.log

Search this file:  Search  
test-topology-1-1459147817/6702/events.log   
Prev First Last Next  
[Download Full File](#)

```
Mon Mar 28 12:20:38 IST 2016,word,13,,[mike]
Mon Mar 28 12:20:38 IST 2016,word,21,,[golda]
Mon Mar 28 12:20:38 IST 2016,word,14,,[golda]
Mon Mar 28 12:20:38 IST 2016,word,15,,[mike]
Mon Mar 28 12:20:38 IST 2016,word,12,,[bertels]
Mon Mar 28 12:20:38 IST 2016,word,15,,[nathan]
Mon Mar 28 12:20:38 IST 2016,word,13,,[golda]
Mon Mar 28 12:20:39 IST 2016,word,12,,[nathan]
Mon Mar 28 12:20:39 IST 2016,word,15,,[jackson]
```

Figure 3: Viewing the logged events.

事件日志中的每一行都包含一个与从特定spout(喷口)/bolt(螺栓)(已逗号分隔的格式)发出的元组相对应的条目。

Timestamp, Component name, Component task-id, MessageId (in case of anchoring), List of emitted values

## 禁用事件日志

可以通过在Storm UI中的拓扑或组件操作下单击“停止调试”，在特定组件或拓扑级别上禁用事件日志。

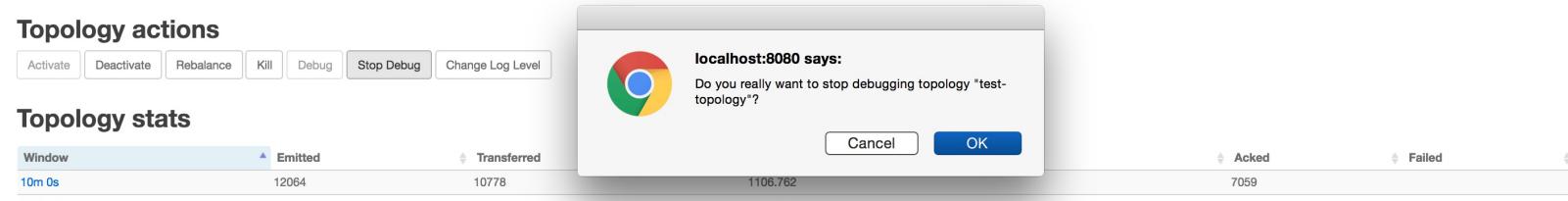


Figure 4: Disable event logging at topology level.

## Configuration

事件记录通过将事件(元组)从每个组件发送到内部事件日志记录工具。默认情况下，Storm不会启动任何事件记录器任务，但可以通过在运行拓扑时设置以下参数(通过在storm.yaml中设置或通过命令传递选项)轻松更改事件记录器任务。

Parameter	Meaning
"topology.eventlogger.executors": 0	No event logger tasks are created (default).
"topology.eventlogger.executors": 1	One event logger task for the topology.
"topology.eventlogger.executors": nil	One event logger task per worker.

## 扩展事件日志

Strom提供了一个“**IEventLogger**”接口，由事件记录器螺栓用于记录事件。这个默认的实现是**FileBasedEventLogger**,它将事件记录到一个事件中。日志文件

(`logs/workers-artifacts/<topology-id>/<worker-port>/events.log`)。可以添加“**IEventLogger**”接口的替代实现来扩展事件记录功能（例如，构建搜索索引或将事件记录到数据库中）`java

```
/** * EventLogger interface for logging the event info to a sink like log file or db * for inspecting the events via UI for debugging. / public interface IEventLogger { /* * Invoked during eventlogger bolt prepare. */ void prepare(Map stormConf, TopologyContext context);
```

```
/**  
 * Invoked when the {@link EventLoggerBolt} receives a tuple from the spouts or bolts that has even  
 *  
 * @param e the event  
 */  
void log(EventInfo e);  
  
/**  
 * Invoked when the event logger bolt is cleaned up  
 */  
void close();  
}  
}```
```

# Storm 与外部系统, 以及其它库的集成

# Storm Kafka Integration

提供核心的 Storm 和 Trident 的 spout 实现，用来从 Apache Kafka 0.8x 版本消费数据。

## Spouts

我们支持 Trident 和 core Storm 的 spout。对于这两种 spout 实现，我们使用 BrokerHosts 接口来跟踪 Kafka broker host partition 映射关系，用 KafkaConfig 来控制 Kafka 相关参数。

## BrokerHosts

为了初始化 Kafka spout/emitter，你需要构造一个 BrokerHosts 标记接口的实例。当前，我们支持以下两种实现方式。

## ZkHosts

如果你想要动态的跟踪 Kafka broker partition 映射关系，你应该使用 ZkHosts。这个类使用 Kafka Zookeeper 实体跟踪 brokerHost->分区映射。你可以调用下面的方法来得到一个实例。

```
java public ZkHosts(String brokerZkStr, String brokerZkPath) public ZkHosts(String brokerZkStr)
```

ZkStr 字符串格式是 ip:port（例如：localhost:2181）。brokerZkPath 是存储所有 topic 和 partition 信息的 zk 根路径。默认情况下，Kafka 使用 /brokers 路径。

默认情况下，broker-partition 映射关系 60s 秒从 Zookeeper 刷新一次。如果你想要改变这个时间，你需要设置 host.refreshFreqSecs 配置。

## StaticHosts

这是一种可替代的实现，broker->partition 信息是静态的。要构造这个类的实例，你需要先构造一个 GlobalPartitionInformation 的实例。

```
Broker brokerForPartition0 = new Broker("localhost");//localhost:9092
Broker brokerForPartition1 = new Broker("localhost", 9092);//localhost:9092 but we specified the
Broker brokerForPartition2 = new Broker("localhost:9092");//localhost:9092 specified as one string
GlobalPartitionInformation partitionInfo = new GlobalPartitionInformation();
partitionInfo.addPartition(0, brokerForPartition0);//mapping from partition 0 to brokerForPartition0
partitionInfo.addPartition(1, brokerForPartition1);//mapping from partition 1 to brokerForPartition1
partitionInfo.addPartition(2, brokerForPartition2);//mapping from partition 2 to brokerForPartition2
StaticHosts hosts = new StaticHosts(partitionInfo);
```

## KafkaConfig

构造一个 KafkaSpout 的实例，第二件事情就是要实例化 KafkaConfig。

```
java public KafkaConfig(BrokerHosts hosts, String topic) public KafkaConfig(BrokerHosts hosts, String
```

BrokerHosts可以通过多个BrokerHosts接口实现.topic就是Kafka topic 的名称.可选择的ClientId就是当前消费的offset存储的zk的路径.

有两个KafkaConfig 继承类正在被使用.

Spoutconfig是KafkaConfig的扩展，它支持Zookeeper 连接信息的其他字段，并且可以控制KafkaSpout的行为.Zkroot就是用来存储消费者offset信息的根路径.id是唯一的，用来标识spout.

```
java public SpoutConfig(BrokerHosts hosts, String topic, String zkRoot, String id); public SpoutConf
```

除此之外， SpoutConfig包含下面这些字段，用来控制KafkaSpout的行为：```java // setting  
for how often to save the current Kafka offset to ZooKeeper public long  
stateUpdateIntervalMs = 2000;

```
// Retry strategy for failed messages
public String failedMsgRetryManagerClass = ExponentialBackoffMsgRetryManager.class.getName();

// Exponential back-off retry settings. These are used by ExponentialBackoffMsgRetryManager for re
// calls OutputCollector.fail(). These come into effect only if ExponentialBackoffMsgRetryManager i
// Initial delay between successive retries
public long retryInitialDelayMs = 0;
public double retryDelayMultiplier = 1.0;

// Maximum delay between successive retries
public long retryDelayMaxMs = 60 * 1000;
// Failed message will be retried infinitely if retryLimit is less than zero.
public int retryLimit = -1;
```

核心KafkaSpout只接口一个SpoutConfig实例

TridentKafkaConfig是KafkaConfig的另外一个扩展.

TridentKafkaEmitter只接受TridentKafkaConfig作为参数.

KafkaConfig类也有一些公共变量来控制你的应用程序的行为。以下是默认值:

```
```java
public int fetchSizeBytes = 1024 * 1024;
public int socketTimeoutMs = 10000;
public int fetchMaxWait = 10000;
public int bufferSizeBytes = 1024 * 1024;
public MultiScheme scheme = new RawMultiScheme();
public boolean ignoreZkOffsets = false;
public long startOffsetTime = kafka.api.OffsetRequest.EarliestTime();
public long maxOffsetBehind = Long.MAX_VALUE;
public boolean useStartOffsetTimeIfOffsetOutOfRange = true;
public int metricsTimeBucketSizeInSecs = 60;
```

除MultiScheme之外，大部分都可以读命名就可以理解。

## MultiScheme

MultiScheme 是一个用来规定 ByteBuffer 如何 Kafka 消费，并转换成一个 storm tuple. 并且会控制 output field 的命名。

```
public Iterable<List<Object>> deserialize(ByteBuffer ser);
public Fields getOutputFields();
```

默认的 RawMultiScheme 接受 ByteBuffer 参数，并返回一个 tuple. 就是将 ByteBuffer 转换成 byte[]. outPutField 的名称是“bytes”。还有可替换的实现，像 SchemeAsMultiScheme 和 KeyValueSchemeAsMultiScheme，他们会将 ByteBuffer 转换成 String.

当然还有个 SchemeAsMultiScheme 的扩展类， MessageMetadataSchemeAsMultiScheme， MessageMetadataSchemeAsMultiScheme 有一个额外的反序列化方法，会接受 ByteBuffer 信息，还会伴随着 Partition 和 offset 信息。

```
public Iterable<List<Object>> deserializeMessageWithMetadata(ByteBuffer message, Partition partition)
```

上面这个方法对于审计/重新处理 Kafka topic 上任意一个点的消息非常有用，保存了每条消息的 partition 和 offset，而不是保留整个消息。

## Failed message retry

FailedMsgRetryManager 是一个定义失败消息的重试策略的接口。默认实现是 ExponentialBackoffMsgRetryManager，它在连续重试之间以指数延迟重试。要使用自定义实现，请将 SpoutConfig.failedMsgRetryManagerClass 设置为完整的实现类名称。下面是接口：

```
java // Spout initialization can go here. This can be called multiple times during lifecycle of a worker.
void prepare(SpoutConfig spoutConfig, Map stormConf);
```

```
// Message corresponding to offset has failed. This method is called only if retryFurther returns true
void failed(Long offset);

// Message corresponding to offset has been acked.
void acked(Long offset);

// Message corresponding to the offset, has been re-emitted and under transit.
void retryStarted(Long offset);

/**
 * The offset of message, which is to be re-emitted. Spout will fetch messages starting from this offset
 * and resend them, except completed messages.
 */
```

```
Long nextFailedMessageToRetry();  
  
/**  
 * @return True if the message corresponding to the offset should be emitted NOW. False otherwise.  
 */  
boolean shouldReEmitMsg(Long offset);  
  
/**  
 * Spout will clean up the state for this offset if false is returned. If retryFurther is set to true  
 * spout will call failed(offset) in next call and acked(offset) otherwise  
 */  
boolean retryFurther(Long offset);  
  
/**  
 * Clear any offsets before kafkaOffset. These offsets are no longer available in kafka.  
 */  
Set<Long> clearOffsetsBefore(Long kafkaOffset);
```

#### #### Version incompatibility

在1.0之前的Storm版本中，MultiScheme方法接受一个`byte []`而不是`ByteBuffer`。 MultiScheme和相关的方案

这意味着，在1.0版及更高版本之前，1.0版的kafka spouts将无法使用。在Storm 1.0及更高版本中运行拓扑时，必须确保

#### ### Examples

#### #### Core Spout

```
```java  
BrokerHosts hosts = new ZkHosts(zkConnString);  
SpoutConfig spoutConfig = new SpoutConfig(hosts, topicName, "/" + topicName, UUID.randomUUID().toString());  
spoutConfig.scheme = new SchemeAsMultiScheme(new StringScheme());  
KafkaSpout kafkaSpout = new KafkaSpout(spoutConfig);
```

## Trident Spout

```
TridentTopology topology = new TridentTopology();  
BrokerHosts zk = new ZkHosts("localhost");  
TridentKafkaConfig spoutConf = new TridentKafkaConfig(zk, "test-topic");  
spoutConf.scheme = new SchemeAsMultiScheme(new StringScheme());  
OpaqueTridentKafkaSpout spout = new OpaqueTridentKafkaSpout(spoutConf);
```

## How KafkaSpout stores offsets of a Kafka topic and recovers in case of failures

如上面的KafkaConfig属性所示，您可以通过设置`KafkaConfig.startOffsetTime`来控制从Kafka topic的哪个端口开始读取，如下所示：

1. `kafka.api.OffsetRequest.EarliestTime()`：从topic初始位置读取消息（例如，从最老的那个消息开始）
2. `kafka.api.OffsetRequestLatestTime()`：从topic尾部开始读取消息（例如，新写入topic的信息）

3. 一个Unix时间戳，从当前 epoch 开始. (例如，可以通过System.currentTimeMillis() )，具体的可以查看Kafka FAQ中的 [How do I accurately get offsets of messages for a certain timestamp using OffsetRequest?](#) .

当topology (拓扑) 运行Kafka Spout，并跟踪读取和发送的offset，并将状态信息存储到zk path `SpoutConfig.zkRoot+ "/" + SpoutConfig.id`. 在故障的情况下，它会从ZooKeeper的最后一次写入偏移中恢复。

**Important:** 新部署topology (拓扑) 时，请确保`SpoutConfig.zkRoot`和`SpoutConfig.id`的设置未被修改，否则spout将无法从ZooKeeper中读取以前的消费者状态信息（即偏移量）导致意外的行为和/或数据丢失，具体取决于您的用例。

这意味着当topology (拓扑) 运行一旦设置`KafkaConfig.startOffsetTime`将不会对 topology (拓扑) 的后续运行产生影响，因为现在 topology (拓扑) 将依赖于ZooKeeper中的消费者状态信息（偏移量）来确定从哪里开始（更多准确地：简历）阅读。如果要强制该端口忽略存储在ZooKeeper中的任何消费者状态信息，则应将参数`KafkaConfig.ignoreZkOffsets` 设置为 true。如果为 true，则如上所述，spout 将始终从`KafkaConfig.startOffsetTime`定义的偏移量开始读取。

## Using storm-kafka with different versions of Kafka

Storm-kafka的Kafka依赖关系在maven中scope 定义为 `provided`，这意味着它不会被作为传递依赖。这允许您使用与Kafka集群兼容的Kafka依赖关系版本。

当使用storm-kafka构建项目时，必须明确地添加Kafka依赖项。例如，要使用针对Scala 2.10构建的Kafka 0.8.1.1，您将在 `pom.xml` 中使用以下依赖关系：

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka_2.10</artifactId>
    <version>0.8.1.1</version>
    <exclusions>
        <exclusion>
            <groupId>org.apache.zookeeper</groupId>
            <artifactId>zookeeper</artifactId>
        </exclusion>
        <exclusion>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

请注意，排除ZooKeeper和log4j依赖关系以防止与Storm的依赖关系发生版本冲突。

您还可以覆盖从maven构建的kafka依赖关系版本，其中包含参数`storm.kafka.version`和`storm.kafka.artifact.id`，例

如`mvn clean install -Dstorm.kafka.artifact.id = kafka_2.11 -Dstorm.kafka.version = 0.9.0.1`

选择kafka依赖版本时，您应该确保 - 1. kafka api与storm-kafka兼容。目前，storm-kafka模块仅支持0.9.x和0.8.x客户端API。如果要使用更高版本，应该使用storm-kafka-client模块替换。 2. 您选择的kafka客户端应与 broker 兼容。例如0.9.x客户端将无法使用0.8.x broker。

## Writing to Kafka as part of your topology

您可以创建一个org.apache.storm.kafka.bolt.KafkaBolt的实例，并将其作为组件附加到topology（拓扑）中，或者如果您使用Trident，则可以使用org.apache.storm.kafka.trident.TridentState，org.apache.storm.kafka.trident.TridentStateFactory和org.apache.storm.kafka.trident.TridentKafkaUpdater。

您需要提供以下2个接口的实现：

### TupleToKafkaMapper and TridentTupleToKafkaMapper

这个接口有下面两个方法：

```
K getKeyFromTuple(Tuple/TridentTuple tuple);
V getMessageFromTuple(Tuple/TridentTuple tuple);
```

顾名思义，这些方法被称为将tuple映射到Kafka key 和Kafka消息。如果您只需要一个字段作为键和一个字段作为值，则可以使用提供的FieldNameBasedTupleToKafkaMapper.java实现。在KafkaBolt中，如果使用默认构造函数构造FieldNameBasedTupleToKafkaMapper，则实现始终会查找字段名称为“key”和“message”的字段，以实现向后兼容性的原因。或者，您也可以使用非默认构造函数指定不同的键和消息字段。在TridentKafkaState中，您必须指定键和消息的字段名称，因为没有默认构造函数。在构造FieldNameBasedTupleToKafkaMapper实例时应该指定这些。

### KafkaTopicSelector and trident KafkaTopicSelector

This interface has only one method

```
java public interface KafkaTopicSelector { String getTopics(Tuple/TridentTuple tuple); }
```

该接口的实现应该返回要发送tuple的密钥/消息映射的topic,您可以返回一个null，该消息将被忽略。如果您有一个静态的topic名称，那么可以使用DefaultTopicSelector.java并在构造函数中设置主题的名称。FieldNameTopicSelector和FieldIndexTopicSelector用于支持决定哪个topic应该从tuple送消息。用户可以在tuple中指定字段名称或字段索引，selector将使用该值作为发布消息的topic名称。当找不到topic名称时，KafkaBolt会将消息写入默认topic。请确保已创建默认topic。

### Specifying Kafka producer properties

`TridentKafkaStateFactory.withProducerProperties()` 来提供 Storm 拓扑中的所有生产属性。有关详细信息，请参阅 <http://kafka.apache.org/documentation.html#newproducerconfigs>“ producer 的重要配置属性”部分。

## Using wildcard kafka topic match

您可以通过添加以下配置来进行通配符 topic 匹配

```
``` Config config = new Config();
config.put("kafka.topic.wildcard.match",true);
```

之后，您可以指定一个通配符 `topic`，以匹配例如点击流。\*记录。这将匹配所有流匹配 `clickstream.my.log`, `clicks`

####Putting it all together

对于 bolt:

```
```java
TopologyBuilder builder = new TopologyBuilder();

Fields fields = new Fields("key", "message");
FixedBatchSpout spout = new FixedBatchSpout(fields, 4,
    new Values("storm", "1"),
    new Values("trident", "1"),
    new Values("needs", "1"),
    new Values("javadoc", "1")
);
spout.setCycle(true);
builder.setSpout("spout", spout, 5);
//set producer properties.
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("acks", "1");
props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

KafkaBolt bolt = new KafkaBolt()
    .withProducerProperties(props)
    .withTopicSelector(new DefaultTopicSelector("test"))
    .withTupleToKafkaMapper(new FieldNameBasedTupleToKafkaMapper());
builder.setBolt("forwardToKafka", bolt, 8).shuffleGrouping("spout");

Config conf = new Config();

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

对于 Trident:

```
Fields fields = new Fields("word", "count");
FixedBatchSpout spout = new FixedBatchSpout(fields, 4,
    new Values("storm", "1"),
    new Values("trident", "1"),
```

```
        new Values("needs", "1"),
        new Values("javadoc", "1")
    );
spout.setCycle(true);

TridentTopology topology = new TridentTopology();
Stream stream = topology.newStream("spout1", spout);

//set producer properties.
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("acks", "1");
props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

TridentKafkaStateFactory stateFactory = new TridentKafkaStateFactory()
    .withProducerProperties(props)
    .withKafkaTopicSelector(new DefaultTopicSelector("test"))
    .withTridentTupleToKafkaMapper(new FieldNameBasedTupleToKafkaMapper("word", "count"));
stream.partitionPersist(stateFactory, fields, new TridentKafkaUpdater(), new Fields());

Config conf = new Config();
StormSubmitter.submitTopology("kafkaTridentTest", conf, topology.build());
```

## Committer Sponsors

P. Taylor Goetz ([ptgoetz@apache.org](mailto:ptgoetz@apache.org))

# Storm Kafka 集成 (0.10.x+)

使用 **kafka-client jar** 进行 **Storm Apache Kafka** 集成

这部分包含新的 Apache Kafka consumer API.

兼容性

Apache Kafka 版本 0.10+

## 写入 Kafka

您可以通过创建 `org.apache.storm.kafka.bolt.KafkaBolt` 实例并将其作为组件附加到您的 `topology`. 如果您使用 `trident`, 您可以通过使用以下对象完成  
`org.apache.storm.kafka.trident.TridentState`,  
`org.apache.storm.kafka.trident.TridentStateFactory` 和  
`org.apache.storm.kafka.trident.TridentKafkaUpdater`.

您需要为以下两个接口提供实现

### `TupleToKafkaMapper` 和 `TridentTupleToKafkaMapper`

这些接口有两个抽象方法:

```
K getKeyFromTuple(Tuple/TridentTuple tuple);
V getMessageFromTuple(Tuple/TridentTuple tuple);
```

顾名思义,这两个方法被调用将tuple映射到Kafka message的key和message本身. 如果你只想要一个字段 作为键和一个字段作为值,那么您可以使用提供的 `FieldNameBasedTupleToKafkaMapper.java` 实现. 在KafkaBolt中,使用默认构造函数构造 `FieldNameBasedTupleToKafkaMapper` 需要一个字段名称为"key"和"message"的字段以实现向后兼容. 或者,您也可以使用非默认构造函数指定不同的键和消息字段. 在使用 `TridentKafkaState` 时你必须明确key和message的字段名称,因为 `TridentKafkaState` 默认的构造函数没有设置参数. 在构造 `FieldNameBasedTupleToKafkaMapper` 的实例时应明确这些.

### `KafkaTopicSelector` 和 `trident KafkaTopicSelector`

这个接口只有一个方法:

```
public interface KafkaTopicSelector {
    String getTopics(Tuple/TridentTuple tuple);
```

}

该接口的实现应该要根据tuple的 key/message 返回相应的Kafka的topic,如果返回 null 则该消息将被忽略掉.如果您只需要一个静态topic名称,那么可以使用 DefaultTopicSelector.java 并在构造函数中设置topic的名称.

FieldNameTopicSelector 和 FieldIndexTopicSelector 用于选择 tuple 要发送到的topic,用户只需要指定tuple中存储 topic名称的字段名称或字段索引即可(即tuple中的某个字段是kafka topic的名称).当topic的名称不存在时, Field\*TopicSelector 会将tuple写入到默认的topic.请确保默认topic已经在kafka中创建并且在Field\*TopicSelector 正确设置.

## 设置 Kafka producer 属性

你可以在 topology 通过调用 KafkaBolt.withProducerProperties() 和 TridentKafkaStateFactory.withProducerProperties() 设置kafka producer的所有属性. Kafka producer配置 选择 "Important configuration properties for the producer" 查看更多详情. 所有的kafka producer配置项的key都在 org.apache.kafka.clients.producer.ProducerConfig类中

## 使用通配符匹配 Kafka topic

通过添加如下属性开启通配符匹配(此功能是为了storm可以动态读取多个kafka topic中的数据,并支持动态发现.看相关功能的实现需求future)

```
Config config = new Config();
config.put("kafka.topic.wildcard.match",true);
```

之后,您可以指定一个通配符topic,例如clickstream.\*.log. 这将匹配 clickstream.my.log,clickstream.cart.log等topic

## bolt 和 Trident 的 Kafka Producer实现

For the bolt :

```
TopologyBuilder builder = new TopologyBuilder();

Fields fields = new Fields("key", "message");
FixedBatchSpout spout = new FixedBatchSpout(fields, 4,
    new Values("storm", "1"),
    new Values("trident", "1"),
    new Values("needs", "1"),
    new Values("javadoc", "1")
);
```

```

spout.setCycle(true);
builder.setSpout("spout", spout, 5);
//set producer properties.
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("acks", "1");
props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

KafkaBolt bolt = new KafkaBolt()
    .withProducerProperties(props)
    .withTopicSelector(new DefaultTopicSelector("test"))
    .withTupleToKafkaMapper(new FieldNameBasedTupleToKafkaMapper());
builder.setBolt("forwardToKafka", bolt, 8).shuffleGrouping("spout");

Config conf = new Config();

StormSubmitter.submitTopology("kafkaboltTest", conf, builder.createTopology());

```

For Trident:

```

Fields fields = new Fields("word", "count");
FixedBatchSpout spout = new FixedBatchSpout(fields, 4,
    new Values("storm", "1"),
    new Values("trident", "1"),
    new Values("needs", "1"),
    new Values("javadoc", "1")
);
spout.setCycle(true);

TridentTopology topology = new TridentTopology();
Stream stream = topology.newStream("spout1", spout);

//set producer properties.
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("acks", "1");
props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

TridentKafkaStateFactory stateFactory = new TridentKafkaStateFactory()
    .withProducerProperties(props)
    .withKafkaTopicSelector(new DefaultTopicSelector("test"))
    .withTridentTupleToKafkaMapper(new FieldNameBasedTupleToKafkaMapper("word", "count"));
stream.partitionPersist(stateFactory, fields, new TridentKafkaUpdater(), new Fields());

Config conf = new Config();
StormSubmitter.submitTopology("kafkaTridentTest", conf, topology.build());

```

## 读取Kafka (Spouts)

配置

spout通过使用 `KafkaSpoutConfig` 类来指定配置. 此类使用 `Builder` 模式, 可以通过调用其中一个 `Builders` 构造函数或通过调用 `KafkaSpoutConfig` 类中的静态方法创建一个 `Builder`. 创建 `builder` 的构造方法或静态方法需要几个键值 (稍后可以更改), 但这是启动一个 `spout` 的所需的最小配置.

`bootstrapServers` 与 Kafka Consumer Property "bootstrap.servers" 相同. 配置项 'topics' 配置的是 `spout` 将消费的 kafka topic. 可以是特定主题名称 (1个或多个) 的集合列表或正则表达式 "Pattern", 它指定 任何与正则表达式匹配的主题都将被消费.

在构造函数的情况下, 您可能还需要指定 `key deserializer` 和 `value deserializer`. 这是为了通过使用 Java 泛型来保证类型安全. 默认值为 "StringDeserializer", 可以通过调用 "setKeyDeserializer" 和 "setValueDeserializer" 进行覆盖. 如果这些设置为 null, 代码将回退到 kafka 属性中设置的内容, 但最好在这里明确, 通过使用 Java 泛型来确保类型安全.

下面是一些需要特别注意的关键配置项.

`setFirstPollOffsetStrategy` 允许你设置从哪里开始消费数据. 这在故障恢复和第一次启动 `spout` 的情况下会被使用. 可选的的值包括:

- `EARLIEST` 无论之前的消费情况如何, `spout` 会从每个 kafka partition 能找到的最早的 offset 开始的读取
- `LATEST` 无论之前的消费情况如何, `spout` 会从每个 kafka partition 当前最新的 offset 开始的读取
- `UNCOMMITTED_EARLIEST` (默认值) `spout` 会从每个 partition 的最后一次提交的 offset 开始读取. 如果 offset 不存在或者过期, 则会依照 `EARLIEST` 进行读取.
- `UNCOMMITTED_LATEST` `spout` 会从每个 partition 的最后一次提交的 offset 开始读取, 如果 offset 不存在或者过期, 则会依照 `LATEST` 进行读取.

`setRecordTranslator` 可以修改 `spout` 如何将 Kafka 消费者 message 转换为 tuple, 以及将该 tuple 发布到哪个 stream 中. 默认情况下, "topic", "partition", "offset", "key" 和 "value" 将被发送到 "default" stream. 如果要将条目根据 topic 输出到不同的 stream 中, Storm 提供了 "ByTopicRecordTranslator". 有关如何使用这些的更多示例, 请参阅下文. `setProp` 可用于设置 kafka 属性. `setGroupId` 可以让您设置 kafka 使用者组属性 "group.id". `setSSLKeystore` 和 `setSSLTruststore` 允许你配置 SSL 认证.

使用举例

API是用java 8 lambda表达式写的. 它也可以用于java7及更低的版本.

## 创建一个简单的不可靠spout

以下将消费kafka中"demo\_topic"的所有消息,并将其发送到MyBolt,其中包括"topic","partition","offset","key","value".

```
final TopologyBuilder tp = new TopologyBuilder();
tp.setSpout("kafka_spout", new KafkaSpout<>(KafkaSpoutConfig.builder("127.0.0.1:" + port, "demo_top
tp.setBolt("bolt", new myBolt()).shuffleGrouping("kafka_spout"));
...
```

## 通配符 Topics

通配符 **topics** 将消费所有符合通配符的**topics**. 在下面的例子中 "topic", "topic\_foo" 和 "topic\_bar" 适配通配符 "topic.\*", 但是 "not\_my\_topic" 并不适配.

```
final TopologyBuilder tp = new TopologyBuilder();
tp.setSpout("kafka_spout", new KafkaSpout<>(KafkaSpoutConfig.builder("127.0.0.1:" + port, Pattern.c
tp.setBolt("bolt", new myBolt()).shuffleGrouping("kafka_spout"));
...
```

## 多个 Streams

这个案例使用 java 8 lambda 表达式.

```
final TopologyBuilder tp = new TopologyBuilder();

//默认情况下,spout 消费但未被match到的topic的消息的"topic","key"和"value"将发送到"STREAM_1"
ByTopicRecordTranslator<String, String> byTopic = new ByTopicRecordTranslator<>(
    (r) -> new Values(r.topic(), r.key(), r.value()),
    new Fields("topic", "key", "value"), "STREAM_1");
//topic_2 所有的消息的 "key" and "value" 将发送到 "STREAM_2"中
byTopic.forTopic("topic_2", (r) -> new Values(r.key(), r.value()), new Fields("key", "value"), "STR

tp.setSpout("kafka_spout", new KafkaSpout<>(KafkaSpoutConfig.builder("127.0.0.1:" + port, "topic_1"
tp.setBolt("bolt", new myBolt()).shuffleGrouping("kafka_spout", "STREAM_1");
tp.setBolt("another", new myOtherBolt()).shuffleGrouping("kafka_spout", "STREAM_2");
...
```

# Trident

```
final TridentTopology tridentTopology = new TridentTopology();
final Stream spoutStream = tridentTopology.newStream("kafkaSpout",
    new KafkaTridentSpoutOpaque<>(KafkaSpoutConfig.builder("127.0.0.1:" + port, Pattern.compile("to"
        .parallelismHint(1)
...

```

Trident不支持多个stream且不支持设置将stream分发到多个output. 并且,如果每个output 的topic的字段不一致会抛出异常而不会继续.

## 自定义 RecordTranslator(高级特性)

在大多数情况下,内置的SimpleRecordTranslator和ByTopicRecordTranslator应该满足您的使用. 如果您遇到需要定制的情况那么这个文档将会描述如何正确地做到这一点,涉及到一些不太常用的类.适用的要点是使用ConsumerRecord并将其转换为可以emitted 的"List ". 难点是如何告诉spout将其发送到指定的stream中. 为此,您将需要返回一个"org.apache.storm.kafka.spout.KafkaTuple"的实例. 这提供了一个方法routedTo,它将说明tuple将要发送到哪个特定stream.

For Example:

```
return new KafkaTuple(1, 2, 3, 4).routedTo("bar");
```

将会使tuple发送到"bar" stream中.

在编写自定义record translators时要小心,因为在Storm spout 中,它需要自我一致. streams方法应该返回这个translator将会尝试发到streams的set列表. 另外, getFieldsFor 应该为每一个 stream 返回一个有效的Fields对象(就是说通过字段名称可以拿到对应的正确的对象). 如果您使用Trident执行此操作,则Fields对象中指定字段的所有值必须在stream名称的List中,否则trident抛出异常. (原文:If you are doing this for Trident a value must be in the List returned by apply for every field in the Fields object for that stream)

## 手动分区控制 (高级特性)

默认情况下,Kafka将自动将partition分配给当前的一组spouts. 它处理很多事情,但在某些情况下,您可能需要手动分配partition.当spout 挂掉并重新启动,但如果处理不正确,可能会导致很多问题. 这可以通过子类化Subscription来处理,我们有几个实现,您可以查看有关如何执行此

操作的示例. `ManualPartitionNamedSubscription`和`ManualPartitionPatternSubscription`. 再次强调, 使用这些或自己实现时请务必注意.

## 使用Maven Shade Plugin构建Uber Jar

Add the following to `REPO_HOME/storm/external/storm-kafka-client/pom.xml`

```
xml <plugin> <groupId>org.apache.maven.plugins</groupId> <artifactId>maven-shade-plugin</artifactId>
```

执行命令生成 uber jar:

```
mvn package -f REPO_HOME/storm/external/storm-kafka-client/pom.xml
```

uber jar 文件会生成在如下目录中:

```
REPO_HOME/storm/external/storm-kafka-client/target/storm-kafka-client-1.0.x.jar
```

## 运行 Storm Topology

复制 `REPO_HOME/storm/external/storm-kafka-client/target/storm-kafka-client-*.jar` 到 `STORM_HOME/extlib`

使用kafka 命令行工具创建topic [test, test1, test2] 并使用 Kafka console producer 向topic 添加数据

执行命令

```
STORM_HOME/bin/storm jar REPO_HOME/storm/external/storm/target/storm-kafka-client-*.jar org.apache.storm.kafka.KafkaSpout
```

开启debug级别日志可以看到每个topic的消息根据设定的stream和设定的shuffle grouping被重定向到相应的spout.

## Using storm-kafka-client with different versions of kafka

Storm-kafka客户端的Kafka依赖关系在maven中被定义为`provided`,这意味着它不会被拉入 作为传递依赖. 这允许您使用与您的kafka集群兼容的Kafka依赖版本.

当使用`storm-kafka-client`构建项目时,必须显式添加Kafka clients依赖关系. 例如,使用Kafka client 0.10.0.0,您将使用以下依赖 `pom.xml`:

```
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka-clients</artifactId>
  <version>0.10.0.0</version>
</dependency>
```

你也可以在使用maven build时通过指定参数`storm.kafka.client.version` 来指定 kafka clients 版本 e.g. `mvn clean install -Dstorm.kafka.client.version=0.10.0.0`

选择kafka client版本时,您应该确保 - 1. kafka api是兼容的. `storm-kafka-client`模块仅支持\*\* 0.10或更新的\*\* kafka客户端API. 对于旧版本, 您可以使用`storm-kafka`模块 (<https://github.com/apache/storm/tree/master/external/storm-kafka>).

2. 您选择的kafka client 应与broker兼容. 例如 0.9.x client 将无法使用 0.8.x broker

## Kafka Spout 性能调整

Kafka spout 提供了两个内置参数来调节其性能. 参数可以通过 [KafkaSpoutConfig](#) 的 `setOffsetCommitPeriodMs` 和 `setMaxUncommittedOffsets`. 方法进行设置

- "offset.commit.period.ms" 控制spout多久向kafka注册一次offset
- "max.uncommitted.offsets" 控制没读取多少条message向kafka注册一次offset

[Kafka consumer config](#) 参数也可能对spout的性能产生影响. 以下Kafka参数可能是spout性能中影响最大的一些参数:

- "fetch.min.bytes"
- "fetch.max.wait.ms"
- [Kafka Consumer](#) Kafka spout 使用 [KafkaSpoutConfig](#) 的 `setPollTimeoutMs`方法设置读取数据的超时时间

根据您的Kafka群集的结构,数据的分布和数据的可用性,这些参数必须正确配置. 请参考关于Kafka参数调整的[Kafka](#)文档.

## kafka spout配置默认值

目前 Kafka spout 有如下默认值,这在[blog post](#)所述的测试环境中表现出了良好的性能

- `poll.timeout.ms = 200`
- `offset.commit.period.ms = 30000 (30s)`
- `max.uncommitted.offsets = 10000000`

## Kafka 自动提交offset模式

如果可靠性对您不重要 - 也就是说,您不关心在失败情况下丢失tuple,并且要消除tuple跟踪的开销,那么您可以使用`AutoCommitMode`运行KafkaSpout.

你需要开启自动提交模式: \* 设置 Config.TOPOLOGY\_ACKERS 为 0; \* 在Kafka consumer 配置中开启 *AutoCommitMode* ;

下面是一个在KafkaSpout中开启AutoCommitMode的例子:

```
KafkaSpoutConfig<String, String> kafkaConf = KafkaSpoutConfig
    .builder(String bootstrapServers, String ... topics)
    .setProp(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "true")
    .setFirstPollOffsetStrategy(FirstPollOffsetStrategy.EARLIEST)
    .build();
```

请注意,由于Kafka消费者定期进行提交*offset*,所以并不是完全符合 *At-Most-Once*.因为在KafkaSpout崩溃时,可能会重读某些*tuple*.

# Storm HBase Integration

Storm/Trident integration for [Apache HBase](#)

## Usage

The main API for interacting with HBase is the `org.apache.storm.hbase.bolt.mapper.HBaseMapper` interface:

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

The `rowKey()` method is straightforward: given a Storm tuple, return a byte array representing the row key.

The `columns()` method defines what will be written to an HBase row. The `ColumnList` class allows you to add both standard HBase columns as well as HBase counter columns.

To add a standard column, use one of the `addColumn()` methods:

```
ColumnList cols = new ColumnList();  
cols.addColumn(this.columnFamily, field.getBytes(), toBytes(tuple.getValueByField(field)));
```

To add a counter column, use one of the `addCounter()` methods:

```
ColumnList cols = new ColumnList();  
cols.addCounter(this.columnFamily, field.getBytes(), toLong(tuple.getValueByField(field)));
```

When the remote HBase is security enabled, a kerberos keytab and the corresponding principal name need to be provided for the storm-hbase connector. Specifically, the Config object passed into the topology should contain {("storm.keytab.file", "\$keytab"), ("storm.kerberos.principal", "\$principal")}. Example:

```
Config config = new Config();  
...  
config.put("storm.keytab.file", "$keytab");  
config.put("storm.kerberos.principal", "$principal");  
StormSubmitter.submitTopology("$topologyName", config, builder.createTopology());
```

## Working with Secure HBASE using delegation tokens.

If your topology is going to interact with secure HBase, your bolts/states needs to be authenticated by HBase. The approach described above requires that all potential worker hosts have "storm.keytab.file" on them. If you have multiple topologies on a cluster , each with different hbase user, you will have to create multiple keytabs and distribute it to all workers. Instead of doing that you could use the following approach:

Your administrator can configure nimbus to automatically get delegation tokens on behalf of the topology submitter user. The nimbus need to start with following configurations:

nimbus.autocredential.plugins.classes : ["org.apache.storm.hbase.security.AutoHBase"]  
nimbus.credential.renewers.classes : ["org.apache.storm.hbase.security.AutoHBase"]  
hbase.keytab.file: "/path/to/keytab/on/nimbus" (This is the keytab of hbase super user that can impersonate other users.) hbase.kerberos.principal: "[superuser@EXAMPLE.com](mailto:superuser@EXAMPLE.com)"  
nimbus.credential.renewers.freq.secs : 518400 (6 days, hbase tokens by default expire every 7 days and can not be renewed, if you have custom settings for hbase.auth.token.max.lifetime in hbase-site.xml than you should ensure this value is atleast 1 hour less then that.)

Your topology configuration should have: topology.auto-credentials :  
["org.apache.storm.hbase.security.AutoHBase"]

If nimbus did not have the above configuration you need to add it and then restart it. Ensure the hbase configuration files(core-site.xml,hdfs-site.xml and hbase-site.xml) and the storm-hbase jar with all the dependencies is present in nimbus's classpath. Nimbus will use the keytab and principal specified in the config to authenticate with HBase. From then on for every topology submission, nimbus will impersonate the topology submitter user and acquire delegation tokens on behalf of the topology submitter user. If topology was started with topology.auto-credentials set to AutoHBase, nimbus will push the delegation tokens to all the workers for your topology and the hbase bolt/state will authenticate with these tokens.

As nimbus is impersonating topology submitter user, you need to ensure the user specified in storm.kerberos.principal has permissions to acquire tokens on behalf of other users. To achieve this you need to follow configuration directions listed on this link

You can read about setting up secure HBase here:<http://hbase.apache.org/book/security.html>.

## SimpleHBaseMapper

`storm-hbase` includes a general purpose `HBaseMapper` implementation called `SimpleHBaseMapper` that can map Storm tuples to both regular HBase columns as well as counter columns.

To use `SimpleHBaseMapper`, you simply tell it which fields to map to which types of columns.

The following code create a `SimpleHBaseMapper` instance that:

1. Uses the `word` tuple value as a row key.
2. Adds a standard HBase column for the tuple field `word`.
3. Adds an HBase counter column for the tuple field `count`.
4. Writes values to the `cf` column family.

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

## HBaseBolt

To use the `HBaseBolt`, construct it with the name of the table to write to, an a `HBaseMapper` implementation:

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

The `HBaseBolt` will delegate to the `mapper` instance to figure out how to persist tuple data to HBase.

## HBaseValueMapper

This class allows you to transform the HBase lookup result into storm Values that will be emitted by the `HBaseLookupBolt`.

```
public interface HBaseValueMapper extends Serializable {
    public List<Values> toTuples(Result result) throws Exception;
```

```
    void declareOutputFields(OutputFieldsDeclarer declarer);  
}
```

The `toTuples` method takes in a HBase `Result` instance and expects a List of `values` instant. Each of the value returned by this function will be emitted by the `HBaseLookupBolt`.

The `declareOutputFields` should be used to declare the outputFields of the `HBaseLookupBolt`.

There is an example implementation in `src/test/java` directory.

## HBaseProjectionCriteria

This class allows you to specify the projection criteria for your HBase Get function. This is optional parameter for the lookupBolt and if you do not specify this instance all the columns will be returned by `HBaseLookupBolt`.

```
public class HBaseProjectionCriteria implements Serializable {  
    public HBaseProjectionCriteria addColumnFamily(String columnFamily);  
    public HBaseProjectionCriteria addColumn(ColumnMetaData column);
```

`addColumnFamily` takes in `columnFamily`. Setting this parameter means all columns for this family will be included in the projection.

`addColumn` takes in a `columnMetaData` instance. Setting this parameter means only this column from the column family will be part of your projection. The following code creates a `projectionCriteria` which specifies a projection criteria that:

1. includes count column from column family cf.
2. includes all columns from column family cf2.

```
HBaseProjectionCriteria projectionCriteria = new HBaseProjectionCriteria()  
.addColumn(new HBaseProjectionCriteria.ColumnMetaData("cf", "count"))  
.addColumnFamily("cf2");
```

## HBaseLookupBolt

To use the `HBaseLookupBolt`, Construct it with the name of the table to write to, an implementation of `HBaseMapper` and an implementation of `HBaseRowToStormValueMapper`. You can optionally specify a `HBaseProjectionCriteria`.

The `HBaseLookupBolt` will use the mapper to get rowKey to lookup for. It will use the

`HBaseProjectionCriteria` to figure out which columns to include in the result and it will leverage the `HBaseRowToStormValueMapper` to get the values to be emitted by the bolt.

You can look at an example topology `LookupWordCount.java` under `src/test/java`.

## Example: Persistent Word Count

A runnable example can be found in the `src/test/java` directory.

### Setup

The following steps assume you are running HBase locally, or there is an `hbase-site.xml` on the classpath pointing to your HBase cluster.

Use the `hbase shell` command to create the schema:

```
> create 'WordCount', 'cf'
```

### Execution

Run the `org.apache.storm.hbase.topology.PersistentWordCount` class (it will run the topology for 10 seconds, then exit).

After (or while) the word count topology is running, run the

`org.apache.storm.hbase.topology.WordCountClient` class to view the counter values stored in HBase. You should see something like to following:

```
Word: 'apple', Count: 6867
Word: 'orange', Count: 6645
Word: 'pineapple', Count: 6954
Word: 'banana', Count: 6787
Word: 'watermelon', Count: 6806
```

For reference, the sample topology is listed below:

```
public class PersistentWordCount {
    private static final String WORD_SPOUT = "WORD_SPOUT";
    private static final String COUNT_BOLT = "COUNT_BOLT";
    private static final String HBASE_BOLT = "HBASE_BOLT";

    public static void main(String[] args) throws Exception {
        Config config = new Config();

        WordSpout spout = new WordSpout();
        WordCounter bolt = new WordCounter();
```

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

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

// wordSpout ==> countBolt ==> HBaseBolt
TopologyBuilder builder = new TopologyBuilder();

builder.setSpout(WORD_SPOUT, spout, 1);
builder.setBolt(COUNT_BOLT, bolt, 1).shuffleGrouping(WORD_SPOUT);
builder.setBolt(HBASE_BOLT, hbase, 1).fieldsGrouping(COUNT_BOLT, new Fields("word"));

if (args.length == 0) {
    LocalCluster cluster = new LocalCluster();
    cluster.submitTopology("test", config, builder.createTopology());
    Thread.sleep(10000);
    cluster.killTopology("test");
    cluster.shutdown();
    System.exit(0);
} else {
    config.setNumWorkers(3);
    StormSubmitter.submitTopology(args[0], config, builder.createTopology());
}
}
```

# Storm HDFS Integration

Storm组件和 HDFS 文件系统交互.

## Usage

以下示例将pipe (“|”) 分隔的文件写入HDFS路径`hdfs://localhost:54310/foo`。 每1000个tuple 之后， 它将同步文件系统， 使该数据对其他HDFS客户端可见。 当它们达到5MB大小时， 它将旋转文件。

```
// use "|" instead of "," for field delimiter
RecordFormat format = new DelimitedRecordFormat()
    .withFieldDelimiter("|");

// sync the filesystem after every 1k tuples
SyncPolicy syncPolicy = new CountSyncPolicy(1000);

// rotate files when they reach 5MB
FileRotationPolicy rotationPolicy = new FileSizeRotationPolicy(5.0f, Units.MB);

FileNameFormat fileNameFormat = new DefaultFileNameFormat()
    .withPath("/foo/");

HdfsBolt bolt = new HdfsBolt()
    .withFsUrl("hdfs://localhost:54310")
    .withFileNameFormat(fileNameFormat)
    .withRecordFormat(format)
    .withRotationPolicy(rotationPolicy)
    .withSyncPolicy(syncPolicy);
```

## Packaging a Topology

当打包你的 topology (拓扑) 代码的时候， 要使用[maven-shade-plugin](#) 插件， 不要使用[maven-assembly-plugin](#)插件.

shade 插件提供了合并 Jar manifest entries 的功能， hadoop client 可以用来做URL scheme 方案.

如果你经历了类似于下面的错误：

```
java.lang.RuntimeException: Error preparing HdfsBolt: No FileSystem for scheme: hdfs
```

这表明你的 topology jar没有正确的打包.

如果你使用maven来创建你的topology jar， 你应该使用下面 `maven-shade-plugin` 配置来创建你的 topology jar:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-shade-plugin</artifactId>
  <version>1.4</version>
  <configuration>
    <createDependencyReducedPom>true</createDependencyReducedPom>
  </configuration>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <goal>shade</goal>
      </goals>
      <configuration>
        <transformers>
          <transformer
            implementation="org.apache.maven.plugins.shade.resource.ServicesResourceTransformer"
            implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer"
            mainClass="com.yammer.storm.hdfs.HDFSConfigurableProcessor"/>
        </transformers>
      </configuration>
    </execution>
  </executions>
</plugin>
```

## Specifying a Hadoop Version

默认情况下， storm-hdfs 使用下面的 Hadoop 依赖.

```
<dependency>
  <groupId>org.apache.hadoop</groupId>
  <artifactId>hadoop-client</artifactId>
  <version>2.2.0</version>
  <exclusions>
    <exclusion>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-log4j12</artifactId>
    </exclusion>
  </exclusions>
</dependency>
<dependency>
  <groupId>org.apache.hadoop</groupId>
  <artifactId>hadoop-hdfs</artifactId>
  <version>2.2.0</version>
  <exclusions>
    <exclusion>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-log4j12</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

如果你使用的Hadoop版本不同，你可以移除storm-hdfs中 Hadoop依赖，并添加你自己的依赖到你的 pom中。

Hadoop客户端版本不兼容，错误如：

```
com.google.protobuf.InvalidProtocolBufferException: Protocol message contained an invalid tag (zero
```

...

...

## Customization

### Record Formats (记录格式化)

记录格式化可以通过提供的`org.apache.storm.hdfs.format.RecordFormat`接口来控制：

```
public interface RecordFormat extends Serializable {  
    byte[] format(Tuple tuple);  
}
```

提供的`org.apache.storm.hdfs.format.DelimitedRecordFormat`实现可以生成如 CSV 和 制表符分隔的文件. T

### File Naming

文件名称可以通过提供的`org.apache.storm.hdfs.format.FileNameFormat`接口来控制：

```
public interface FileNameFormat extends Serializable {  
    void prepare(Map conf, TopologyContext topologyContext);  
    String getName(long rotation, long timeStamp);  
    String getPath();  
}
```

提供的`org.apache.storm.hdfs.format.DefaultFileNameFormat` 创建的文件名称格式如下：

```
{prefix}{componentId}-{taskId}-{rotationNum}-{timestamp}{extension}
```

例如：

```
MyBolt-5-7-1390579837830.txt
```

默认情况下，前缀是空的，扩展标识是".txt".

### Sync Policies

同步策略允许你将 buffered data 缓冲到底层文件系统（从而client可以读取数据），通过实

现org.apache.storm.hdfs.sync.SyncPolicy 接口：

```
public interface SyncPolicy extends Serializable {  
    boolean mark(Tuple tuple, long offset);  
    void reset();  
}
```

HdfsBolt 会为每个要处理的 tuple 调用 mark()方法.返回 true 会触发 HdfsBolt 执行同步/刷新，之后会调用 reset()方法.

org.apache.storm.hdfs.sync.CountSyncPolicy 类可以简单的触发同步，当一定数量的tuple执行完成后.

## File Rotation Policies

类似于同步策略,文件反转策略允许你通过 org.apache.storm.hdfs.rotation.FileRotation 接口来控制数据文件反转.

```
public interface FileRotationPolicy extends Serializable {  
    boolean mark(Tuple tuple, long offset);  
    void reset();  
}
```

org.apache.storm.hdfs.rotation.FileSizeRotationPolicy 实现允许数据文件达到指定的文件大小后，触发文件反转.

```
FileRotationPolicy rotationPolicy = new FileSizeRotationPolicy(5.0f, Units.MB);
```

## File Rotation Actions

HDFS bolt 和 Trident State 实现允许你注册任意数量的 RotationActionS. RotationActionS 要做的就是提供一个hook，当文件反转后执行一些操作。例如，移动一个文件到不同的路径下，或者重命名.

```
public interface RotationAction extends Serializable {  
    void execute(FileSystem fileSystem, Path filePath) throws IOException;  
}
```

Storm-HDFS 包括一个简单的操作，反转后移动一个文件:

```
public class MoveFileAction implements RotationAction {  
    private static final Logger LOG = LoggerFactory.getLogger(MoveFileAction.class);  
  
    private String destination;
```

```

public MoveFileAction withDestination(String destDir){
    destination = destDir;
    return this;
}

@Override
public void execute(FileSystem fileSystem, Path filePath) throws IOException {
    Path destPath = new Path(destination, filePath.getName());
    LOG.info("Moving file {} to {}", filePath, destPath);
    boolean success = fileSystem.rename(filePath, destPath);
    return;
}
}

```

如果你使用 Trident，并且是有序的文件，你可以像下面这样使用：

```

HdfsState.Options seqOpts = new HdfsState.SequenceFileOptions()
    .withFileNameFormat(fileNameFormat)
    .withSequenceFormat(new DefaultSequenceFormat("key", "data"))
    .withRotationPolicy(rotationPolicy)
    .withFsUrl("hdfs://localhost:54310")
    .addRotationAction(new MoveFileAction().withDestination("/dest2/"));

```

## Support for HDFS Sequence Files

`org.apache.storm.hdfs.bolt.SequenceFileBolt`类允许你写入storm data 到连续的HDFS文件中：

```

// sync the filesystem after every 1k tuples
SyncPolicy syncPolicy = new CountSyncPolicy(1000);

// rotate files when they reach 5MB
FileRotationPolicy rotationPolicy = new FileSizeRotationPolicy(5.0f, Units.MB);

FileNameFormat fileNameFormat = new DefaultFileNameFormat()
    .withExtension(".seq")
    .withPath("/data/");

// create sequence format instance.
DefaultSequenceFormat format = new DefaultSequenceFormat("timestamp", "sentence");

SequenceFileBolt bolt = new SequenceFileBolt()
    .withFsUrl("hdfs://localhost:54310")
    .withFileNameFormat(fileNameFormat)
    .withSequenceFormat(format)
    .withRotationPolicy(rotationPolicy)
    .withSyncPolicy(syncPolicy)
    .withCompressionType(SequenceFile.CompressionType.RECORD)
    .withCompressionCodec("deflate");

```

`SequenceFileBolt` 需要你提供一个 `org.apache.storm.hdfs.bolt.format.SequenceFormat`，用来映射 tuples到 key/value pairs。

```
public interface SequenceFormat extends Serializable {
```

```

    Class keyClass();
    Class valueClass();

    Writable key(Tuple tuple);
    Writable value(Tuple tuple);
}

```

## Trident API

storm-hdfs 还包括一个 Trident `state` 实现，用于写入数据到HDFS，API类似于 bolts.

```

Fields hdfsFields = new Fields("field1", "field2");

FileNameFormat fileNameFormat = new DefaultFileNameFormat()
    .withPath("/trident")
    .withPrefix("trident")
    .withExtension(".txt");

RecordFormat recordFormat = new DelimitedRecordFormat()
    .withFields(hdfsFields);

FileRotationPolicy rotationPolicy = new FileSizeRotationPolicy(5.0f, FileSizeRotationPolicy

HdfsState.Options options = new HdfsState.HdfsFileOptions()
    .withFileNameFormat(fileNameFormat)
    .withRecordFormat(recordFormat)
    .withRotationPolicy(rotationPolicy)
    .withFsUrl("hdfs://localhost:54310");

StateFactory factory = new HdfsStateFactory().withOptions(options);

TridentState state = stream
    .partitionPersist(factory, hdfsFields, new HdfsUpdater(), new Fields());

```

要使用序列文件 `State` 实现，请使用 `HdfsState.SequenceFileOptions`:

```

HdfsState.Options seqOpts = new HdfsState.SequenceFileOptions()
    .withFileNameFormat(fileNameFormat)
    .withSequenceFormat(new DefaultSequenceFormat("key", "data"))
    .withRotationPolicy(rotationPolicy)
    .withFsUrl("hdfs://localhost:54310")
    .addRotationAction(new MoveFileAction().toDestination("/dest2/"));

```

## Working with Secure HDFS

如果您的 topology (拓扑) 将与安全的HDFS进行交互，则您的 bolts/states 需要通过 NameNode 进行身份验证。我们 目前有2个选项支持:

### Using HDFS delegation tokens

您的管理员可以配置nimbus来代表拓扑提交者用户自动获取授权令牌。 nimbus需要从以下配置开始：

nimbus.autocredential.plugins.classes :

```
["org.apache.storm.hdfs.common.security.AutoHDFS"] nimbus.credential.renewers.classes  
: ["org.apache.storm.hdfs.common.security.AutoHDFS"] hdfs.keytab.file:  
"/path/to/keytab/on/nimbus" (hdfs 超级管理员可以代理其他用户.) hdfs.kerberos.principal:  
"superuser@EXAMPLE.com" nimbus.credential.renewers.freq.secs : 82800 (23 小时, hdfs  
tokens 需要每24个小时更新一次.) topology.hdfs.uri:"hdfs://host:port" (可选的配置, 默认情  
况下, 我们会在core-site.xml 文件中指定 "fs.defaultFS" 属性)
```

你的topology 配置应该包括： topology.auto-credentials :

```
["org.apache.storm.hdfs.common.security.AutoHDFS"]
```

如果nimbus没有上述配置, 您需要添加它, 然后重新启动它。确保hadoop配置文件（core-site.xml和hdfs-site.xml）以及具有所有依赖项的storm-hdfs jar都存在于nimbus的类路径中。Nimbus将使用配置文件中指定的 keytab 和主体对 Namenode 进行身份验证。从那时起每一个 topology 提交, nimbus将模拟拓扑提交者用户并代表代理令牌 topology 提交者用户。如果通过将topology.auto-credentials设置为AutoHDFS启动 topology（拓扑）, nimbus将推送将所有的工作人员的代理令牌用于您的 topology（拓扑）, 并且hdfs bolt / state将使用namenode进行身份验证这些令牌。

由于nimbus模拟topology（拓扑）提交者用户, 您需要确保hdfs.kerberos.principal中指定的用户 具有代表其他用户获取令牌的权限。要实现这一点, 您需要遵循配置指导 列在此链接上: <http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/Superusers.html>

你可以看这里如何配置安全的HDFS: <http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/SecureMode.html>.

## Using keytabs on all worker hosts

如果您已将hdfs用户的 keytab 文件分发给所有潜在的worker , 那么可以使用此方法。你应该指定一个 使用HdfsBolt / State.withconfigKey（“somekey”）方法的hdfs配置密钥, 该密钥的值映射应具有以下2个属性:

hdfs.keytab.file: "/path/to/keytab/" hdfs.kerberos.principal: "[user@EXAMPLE.com](mailto:user@EXAMPLE.com)"

在workers 上, bolt/Ttrident-staet code 将使用配置中提供的主体的keytab文件进行认证 Namenode。这种方法很危险, 因为您需要确保所有 worker 的keytab文件位于同一位置, 您需要 在集群中启动新主机时记住这一点.

# Storm Hive 集成

Hive 提供了 streaming API, 它允许将数据连续地写入 Hive. 传入的数据可以用小批量 record 的方式连续提交到现有的 Hive partition 或 table 中. 一旦提交了数据, 它就可以立即显示给所有的 hive 查询. 有关 Hive Streaming API 的更多信息请参阅

<https://cwiki.apache.org/confluence/display/Hive/Streaming+Data+Ingest>

在 Hive Streaming API 的帮助下, HiveBolt 和 HiveState 允许用户将 Storm 中的数据直接传输到 Hive 中. 要使用 Hive streaming API, 用户需要创建一个使用了 ORC 格式的 bucketed table. 如下所示

```
create table test_table ( id INT, name STRING, phone STRING, street STRING) partitioned by (city
```

...  
...

## HiveBolt (org.apache.storm.hive.bolt.HiveBolt)

HiveBolt 控制 tuples 直接流入到 Hive 中. 使用 Hive 事务写入 Tuples. HiveBolt 将流式传输的分区可以创建或预先创建, 或者也可用 HiveBolt 来创建它们, 如果它们不存在的话.

```
DelimitedRecordHiveMapper mapper = new DelimitedRecordHiveMapper()
    .withColumnFields(new Fields(colNames));
HiveOptions hiveOptions = new HiveOptions(metaStoreURI, dbName, tblName, mapper);
HiveBolt hiveBolt = new HiveBolt(hiveOptions);
```

## RecordHiveMapper

该 class 将 Tuple 的字段名映射到 Hive table 的列名.

- DelimitedRecordHiveMapper  
(org.apache.storm.hive.bolt.mapper.DelimitedRecordHiveMapper)
- JsonRecordHiveMapper (org.apache.storm.hive.bolt.mapper.JsonRecordHiveMapper)

```
DelimitedRecordHiveMapper mapper = new DelimitedRecordHiveMapper()
    .withColumnFields(new Fields(colNames))
    .withPartitionFields(new Fields(partNames));
or
DelimitedRecordHiveMapper mapper = new DelimitedRecordHiveMapper()
    .withColumnFields(new Fields(colNames))
    .withTimeAsPartitionField("YYYY/MM/DD");
```

Arg (参数)	Description (描述)	Type (类型)
	tuple 中要被映射到 table 列名的字段名	

withColumnFields	称	Fields (必需的)
withPartitionFields	tuple 中要被映射到 hive table partition 的字段名称	Fields
withTimeAsPartitionField	用户可以使用系统时间作为 hive table 的 partition	String . Date format

## HiveOptions (org.apache.storm.hive.common.HiveOptions)

HiveBolt 将 HiveOptions 作为一个构造参数.

```
HiveOptions hiveOptions = new HiveOptions(metaStoreURI, dbName, tblName, mapper)
    .withTxnsPerBatch(10)
    .withBatchSize(1000)
    .withIdleTimeout(10)
```

### HiveOptions 参数

Arg (参数)	Description (描述)	Type (类型)
metaStoreURI	hive meta store URI (可以在 <code>hive-site.xml</code> 中找到)	String (必需的)
dbName	数据库名	String (必需的)
tblName	表名	String (必需的)
mapper	Mapper class, 映射 Tuple 的字段名称到 Table 的列名称	DelimitedRecordHiveMapper 或 JsonRecordHiveMapper (必需的)
withTxnsPerBatch	Hive 向 HiveBolt 的流客户端授予一批事务 而不是单个事务. 此设置配置每个事务批处理所需的事务数. 来自单个批次中所有事务的数据最终在单个文件中. Flume 将在批处理中的每个事务中写入最大的 <code>batchSize</code> 事件. 与 <code>batchSize</code> 配合使用的设置可以控制每个文件的大小. 请注意, 最终 Hive 将透明地将这些文件压缩成较大	Integer . 默认 100

	的文件.	
withMaxOpenConnections	只允许这个数量的 <code>open connections</code> . 如果超过该数量, 则最近最少使用的 connection 将被 <code>closed</code> .	Integer. 默认 100
withBatchSize	在单个 Hive 事务中写入 Hive 的最大事件数	Integer. 默认 15000
withCallTimeout	(In milliseconds) 针对 Hive & HDFS I/O operations 的超时, 例如 <code>openTxn</code> , <code>write</code> , <code>commit</code> , <code>abort</code> .	Integer. 默认 10000
withHeartBeatInterval	(In seconds) Interval between consecutive heartbeats sent to Hive to keep unused transactions from expiring. Set this value to 0 to disable heartbeats.	Integer. 默认 240
withAutoCreatePartitions	HiveBolt 将自动创建必要的 Hive partition 以流式传输.	Boolean. 默认 true
withKerberosPrincipal	Kerberos user principal 用于安全的访问 Hive	String
withKerberosKeytab	Kerberos keytab 用户安全的访问 Hive	String
withTickTupleInterval	(In seconds) 如果 $> 0$ , 则 Hive Bolt 将定期刷新事务批次. 建议启用此功能, 以避免在等待批次阻塞时出现元组超时.	Integer. 默认 0

## HiveState (org.apache.storm.hive.trident.HiveTrident)

Hive Trident state 也遵循 HiveBolt 类似的模式, 它以 `HiveOptions` 作为参数.

```
DelimitedRecordHiveMapper mapper = new DelimitedRecordHiveMapper()
    .withColumnFields(new Fields(colNames))
    .withTimeAsPartitionField("YYYY/MM/DD");
```

```
HiveOptions hiveOptions = new HiveOptions(metaStoreURI,dbName,tblName,mapper)
        .withTxnsPerBatch(10)
        .withBatchSize(1000)
        .withIdleTimeout(10)

StateFactory factory = new HiveStateFactory().withOptions(hiveOptions);
TridentState state = stream.partitionPersist(factory, hiveFields, new HiveUpdater(), new Fields()
```

# Storm Solr 集成

针对 Apache Solr 的 Storm 和 Trident 集成. 该软件包包括一个 bolt 和 trident state, 它们可以使 Storm topology 将 storm tuples 的内容索引到 Solr collections.

## Index Storm tuples 到 Solr collection 中

bolt 和 trident state 使用一个提供的 mappers (映射器) 来构建一个 `SolrRequest` 对象, 负责对 Solr 进行更新调用, 从而更新指定的 collection 的 index.

### 使用示例

在本节中, 我们将提供一些关于如何构建 Storm 和 Trident topologies 来索引 Solr 的简单代码片段. 在随后的章节中, 我们详细介绍了 Storm Solr 集成的两个关键组件, `SolrUpdateBolt` 和 `Mappers`, `SolrFieldsMapper` 和 `SolrJsonMapper`.

## Storm Bolt 与 JSON Mapper 和 Count Based Commit Strategy

```
new SolrUpdateBolt(solrConfig, solrMapper, solrCommitStgy)

// zkHostString for Solr 'gettingstarted' example
SolrConfig solrConfig = new SolrConfig("127.0.0.1:9983");

// JSON Mapper used to generate 'SolrRequest' requests to update the "gettingstarted" Solr coll
SolrMapper solrMapper = new SolrJsonMapper.Builder("gettingstarted", "JSON").build();

// Acks every other five tuples. Setting to null acks every tuple
SolrCommitStrategy solrCommitStgy = new CountBasedCommit(5);
```

## Trident Topology 与 Fields Mapper

```
new SolrStateFactory(solrConfig, solrMapper);

// zkHostString for Solr 'gettingstarted' example
SolrConfig solrConfig = new SolrConfig("127.0.0.1:9983");

/* Solr Fields Mapper used to generate 'SolrRequest' requests to update the "gettingstarted" So
SolrMapper solrMapper = new SolrFieldsMapper.Builder(schemaBuilder, "gettingstarted").build();

// builds the Schema object from the JSON representation of the schema as returned by the URL h
SchemaBuilder schemaBuilder = new RestJsonSchemaBuilder("localhost", "8983", "gettingstarted")
```

## SolrUpdateBolt

`SolrUpdateBolt` 让 tuples 直接流入到 Apache Solr 中. 该 Solr index 使用 `SolrRequest` 请求来更

新. 该 `SolrUpdateBolt` 可以通过实现 `SolrConfig`, `SolrMapper` 和可选的 `SolrCommitStrategy` 方法来配置它.

使用 `SolrMapper` 实现中定义的策略从 `tuples` 中提取 `stream` 到 Solr 的数据.

`SolrRequest` 可以用来发送每个 `tuple`, 也可以根据 `SolrCommitStrategy` 实现中定义的策略进行发送. 如果 `SolrCommitStrategy` 已经到位并且批处理中的一个元组失败, 则该 `batch` 不会被提交, 并且该 `bathc` 中的所有 `tuple` 都标记为 `Fail`, 并重试. 另一方面, 如果所有的 `tuple` 都成功, 则 `SolrRequest` 被提交, 所有 `tuple` 都被成功地 `acked` (确认) .

`SolrConfig` 是包含 Solr 配置的 `class`, 可用于 Storm Solr bolt. bolt 中需要的任何配置都应该放在这个 `class` 中.

## SolrMapper

`SorlMapper` 实现定义了从 `tuple` 中提取信息的策略. `public method toSolrRequest` 接收一个 `tuple` 或一组 `tuple`, 并返回一个用于更新 Solr 索引的 `SolrRequest` 对象.

## SolrJsonMapper

`SolrJsonMapper`` 创建一个 Solr 更新请求, 该请求被发送到由 Solr 定义的作为 JSON 格式请求的 URL endpoint.

要创建一个 `SolrJsonMapper`, 客户端必须指定要更新的 `collection` 的名称以及包含用于更新 Solr 索引的 JSON 对象的 `tuple field` (元组字段). 如果 `tuple` 不包含指定的字段, 则在调用方法 `toSolrRequest` 时抛出 `SolrMapperException`. 如果该字段存在, 则其值可以是 JSON 格式的内容的 `String`, 或将被序列化为 JSON 的 `Java` 对象.

下面的代码片段, 说明如何创建一个 `SolrJsonMapper` 对象, 以更新 `gettingstarted` Solr collection, 并使用名称为 `JSON` 的 `tuple` 字段中声明的 JSON 内容

```
SolrMapper solrMapper = new SolrJsonMapper.Builder("gettingstarted", "JSON").build();
```

## SolrFieldsMapper

`SolrFieldsMapper` 创建一个 Solr 更新请求, 该请求被发送到处理 `SolrInputDocument` 对象的更新的 Solr URL endpoint.

要创建一个 `SolrFieldsMapper`, 客户端必须指定要更新的集合的名称以及 `SolrSchemaBuilder`.

Solr 的 Schema 用于提取有关 Solr schema 字段和相应类型的信息。该 metadata 用于从 tuples 中获取信息。只有匹配静态或动态 Solr 字段的 tuple 字段才会添加到文档中。与 schema 不匹配的 tuple 字段不添加到准备进行索引的 SolrInputDocument 中。针对不匹配 schema 的 tuple 字段打印出 debug log message，因此不进行索引。

SolrFieldsMapper 支持多 value 的字段。多 value 的 tuple 字段必须被 tokenized（分词）。默认的 token（词元）是 |。可以通过调用作为 SolrFieldsMapper.Builder builder class 中的方法 org.apache.storm.solr.mapper.SolrFieldsMapper.Builder.setMultiValueFieldToken 来指定任意的 token（词元）。

下面的代码片段演示了如何去创建一个 SolrFieldsMapper 对象以更新 gettingstarted 的 Solr collection。该多个 value 的字段使用 % 而不是默认的 | 来拆分每个 value。要使用默认的 token 您可以省略对 setMultiValueFieldToken 方法的调用。

```
new SolrFieldsMapper.Builder()
    new RestJsonSchemaBuilder("localhost", "8983", "gettingstarted"), "gettingstarted")
        .setMultiValueFieldToken("%").build();
```

## 构建并且运行附带的示例

为了能够运行这些示例，您必须首先在 storm-solr 包中构建 java 代码，然后生成具有所有依赖项的 uber jar。

## 构建 Storm Apache Solr 集成的代码

```
mvn clean install -f REPO_HOME/storm/external/storm-solr/pom.xml
```

## 使用 Maven Shade 插件来构建 Uber Jar

添加以下配置到 REPO\_HOME/storm/external/storm-solr/pom.xml 中：

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-shade-plugin</artifactId>
  <version>2.4.1</version>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <goal>shade</goal>
      </goals>
      <configuration>
        <transformers>
          <transformer implementation="org.apache.maven.plugins.shade.resource.ManifestR
            <mainClass>org.apache.storm.solr.topology.SolrJsonTopology</mainClass>
```

```
</transformer>
</transformers>
</configuration>
</execution>
</executions>
</plugin>
```

通过运行命令以创建 **uber jar**:

```
mvn package -f REPO_HOME/storm/external/storm-solr/pom.xml
```

这将创建一个与以下模式匹配的名称和位置的 **uber jar** 文件:

```
REPO_HOME/storm/external/storm/target/storm-solr-0.11.0-SNAPSHOT.jar
```

## 运行示例

复制文件 `REPO_HOME/storm/external/storm-solr/target/storm-solr-0.11.0-SNAPSHOT.jar` 到 `STORM_HOME/extlib`

**The code examples provided require that you first run the [Solr gettingstarted example](#)**

## 运行 **Storm Topology**

```
STORM_HOME/bin/storm jar REPO_HOME/storm/external/storm-solr/target/storm-solr-0.11.0-SNAPSHOT-test
```

```
STORM_HOME/bin/storm jar REPO_HOME/storm/external/storm-solr/target/storm-solr-0.11.0-SNAPSHOT-test
```

## 运行 **Trident Topology**

```
STORM_HOME/bin/storm jar REPO_HOME/storm/external/storm-solr/target/storm-solr-0.11.0-SNAPSHOT-test
```

```
STORM_HOME/bin/storm jar REPO_HOME/storm/external/storm-solr/target/storm-solr-0.11.0-SNAPSHOT-test
```

## 验证结果

上述的 **Storm** 和 **Trident topologies** 索引了 Solr `getstarted` collect, 其对象具有以下的 'id' 模式:

\*`id_fields_test_val`\* 用于 `SolrFieldsTopology` 和 `SolrFieldsTridentTopology`

\*`json_test_val`\* 用于 `SolrJsonTopology` 和 `SolrJsonTridentTopology`

查询这些模式的 Solr，您将看到由 Storm Apache Solr 集成索引的值：

```
curl -X GET -H "Content-type:application/json" -H "Accept:application/json" http://localhost:8983/s
curl -X GET -H "Content-type: application/json" -H "Accept: application/json" http://localhost:8983
```

您还可以通过打开 Apache Solr UI 并在查询页面中的  文本框中粘贴 **id** 模式来查看结果

```
http://localhost:8983/solr/#/gettingstarted_shard1_replica2/query
```

# Storm Cassandra 集成

## Apache Cassandra 的 Bolt API 实现

这个库提供了 Apache Cassandra 之上的核心 storm bolt . 提供简单的 DSL 来 map storm *Tuple* 到 Cassandra Query Language *Statement* (Cassandra 查询语言 *Statement*) .

## Configuration (配置)

以下属性可能会传递给 storm 配置.

Property name (属性名称)	Description (描述)	Default (默认)
cassandra.keyspace	-	
cassandra.nodes	-	{"localhost"}
cassandra.username	-	-
cassandra.password	-	-
cassandra.port	-	9092
cassandra.output.consistencyLevel	-	ONE
cassandra.batch.size.rows	-	100
cassandra.retryPolicy	-	DefaultRetryPolicy
cassandra.reconnectionPolicy.baseDelayMs	-	100 (ms)
cassandra.reconnectionPolicy.maxDelayMs	-	60000 (ms)

## CassandraWriterBolt

### Static import

```
import static org.apache.storm.cassandra.DynamicStatementBuilder.*
```

## Insert Query Builder (插入查询生成器)

**Insert query including only the specified tuple fields** (插入仅包含指定 tuple 字段的查询) .

```
new CassandraWriterBolt(  
    async(  
        simpleQuery("INSERT INTO album (title,year,performer,genre,tracks) VALUES (?, ?, ?, ?, ?,  
            .with(  
                fields("title", "year", "performer", "genre", "tracks")  
            )  
        )  
    );
```

**Insert query including all tuple fields** (插入包含所有 tuple 字段的查询) .

```
new CassandraWriterBolt(  
    async(  
        simpleQuery("INSERT INTO album (title,year,performer,genre,tracks) VALUES (?, ?, ?, ?, ?,  
            .with( all() )  
        )  
    );
```

**Insert multiple queries from one input tuple** (从一个 input tuple 插入多个查询) .

```
new CassandraWriterBolt(  
    async(  
        simpleQuery("INSERT INTO titles_per_album (title,year,performer,genre,tracks) VALUES (?)  
        simpleQuery("INSERT INTO titles_per_performer (title,year,performer,genre,tracks) VALUE  
    )  
);
```

**Insert query using QueryBuilder** (使用 **QueryBuilder** 插入查询)

```
new CassandraWriterBolt(  
    async(  
        simpleQuery("INSERT INTO album (title,year,perfomer,genre,tracks) VALUES (?, ?, ?, ?, ?, ?  
            .with(all()))  
        )  
    );
```

**Insert query with static bound query** (使用 **static bound query** 插入 查询)

```
new CassandraWriterBolt(  
    async(  
        boundQuery("INSERT INTO album (title,year,performer,genre,tracks) VALUES (?, ?, ?, ?, ?, ?)  
            .bind(all());  
    )  
);
```

## Insert query with static bound query using named setters and aliases (使用 named setters 和 aliases 插入带有 static bound query 的查询)

```
new CassandraWriterBolt(  
    async(  
        boundQuery("INSERT INTO album (title,year,performer,genre,tracks) VALUES (:ti, :ye, :pe  
            .bind(  
                field("ti").as("title"),  
                field("ye").as("year")),  
                field("pe").as("performer")),  
                field("ge").as("genre")),  
                field("tr").as("tracks"))  
            .byNamedSetters()  
    )  
);
```

## Insert query with bound statement load from storm configuration (从 storm 配置插入 bound statement load 的查询)

```
new CassandraWriterBolt(  
    boundQuery(named("insertIntoAlbum"))  
    .bind(all()));
```

## Insert query with bound statement load from tuple field (从 tuple 字段插入 bound statement load 的查询)

```
new CassandraWriterBolt(  
    boundQuery(namedByField("cql"))  
    .bind(all()));
```

## Insert query with batch statement (使用 batch 语句插入查询)

```
// Logged  
new CassandraWriterBolt(loggedBatch(  
    simpleQuery("INSERT INTO titles_per_album (title,year,performer,genre,tracks) VALUES (?)  
    simpleQuery("INSERT INTO titles_per_performer (title,year,performer,genre,tracks) VALUE
```

```
        );
    // UnLogged
    new CassandraWriterBolt(unLoggedBatch(
        simpleQuery("INSERT INTO titles_per_album (title,year,performer,genre,tracks) VALUES (?)")
        simpleQuery("INSERT INTO titles_per_performer (title,year,performer,genre,tracks) VALUES (?)")
    );
}
```

## 如何处理 **query execution results** (查询执行结果)

*ExecutionResultHandler* 接口可用于自定义 *execution result* (执行结果) 应如何处理.

```
public interface ExecutionResultHandler extends Serializable {
    void onQueryValidationException(QueryValidationException e, OutputCollector collector, Tuple tuple);
    void onReadTimeoutException(ReadTimeoutException e, OutputCollector collector, Tuple tuple);
    void onWriteTimeoutException(WriteTimeoutException e, OutputCollector collector, Tuple tuple);
    void onUnavailableException(UnavailableException e, OutputCollector collector, Tuple tuple);
    void onQuerySuccess(OutputCollector collector, Tuple tuple);
}
```

默认情况下, *CassandraBolt* 将在所有的 *Cassandra Exception* 中 fails (失败) 一个 tuple (请参阅 [BaseExecutionResultHandler](#)) .

```
new CassandraWriterBolt(insertInto("album").values(with(all()).build())
    .withResultHandler(new MyCustomResultHandler()));
```

## Declare Output fields (声明输出字段)

*CassandraBolt* 可以声明 *declare output fields / stream output fields* (输出字段/流输出字段). 例如, 这可以用于在 *error* (错误) 或者 *chain queries* (链式查询) 上 *remit* (传递) 一个 *new tuple* (新的元组) .

```
new CassandraWriterBolt(insertInto("album").values(withFields(all()).build())
    .withResultHandler(new EmitOnDriverExceptionResultHandler()));
    .withStreamOutputFields("stream_error", new Fields("message"));

public static class EmitOnDriverExceptionResultHandler extends BaseExecutionResultHandler {
    @Override
    protected void onDriverException(DriverException e, OutputCollector collector, Tuple tuple)
        LOG.error("An error occurred while executing cassandra statement", e);
        collector.emit("stream_error", new Values(e.getMessage()));
        collector.ack(tuple);
}
```

```
}
```

```
III
```

```
>
```

## Murmur3FieldGrouping

[Murmur3StreamGrouping](#) 可以用来优化 `cassandra writes` (`cassandra` 的写入) . 根据 specified row partition keys (指定的行分区键) , 该 stream 在 bolt 的 task 之间进行 partitioned (分区) .

```
CassandraWriterBolt bolt = new CassandraWriterBolt(  
    insertInto("album")  
    .values(  
        with(fields("title", "year", "performer", "genre", "tracks")  
        ).build());  
builder.setBolt("BOLT_WRITER", bolt, 4)  
    .customGrouping("spout", new Murmur3StreamGrouping("title"))
```

## Trident API 支持

storm-cassandra 支持 用于将 data `inserting` (插入) Cassandra 的 Trident state API .

```
java CassandraState.Options options = new CassandraState.Options(new CassandraContext()); CQLStateme
```

以下 state API 用于从 Cassandra `querying` (查询) 数据.

```
java CassandraState.Options options = new CassandraState.Options(new CassandraContext()); CQLStateme
```

# Storm JDBC 集成

Storm/Trident集成JDBC. 该包中包含的核心bolts 和 trident states， 允许storm topology把 storm tuples插入数据库表中或者执行数据库查询， 并且丰富了storm topology 中的tuples.

注： 在下面的示例中， 我们使用com.google.common.collect.Lists和 com.google.common.collect.Maps。

## Inserting into a database. 插入数据库.

该包的中bolt 和 trident state可以将数据插入到数据库表中绑定到单个表。

### ConnectionProvider

由不同的连接池机制实现的接口 `org.apache.storm.jdbc.common.ConnectionProvider`

```
java public interface ConnectionProvider extends Serializable { /** * method must be idempotent. */ void prepare();
```

```
/**  
 *  
 * @return a DB connection over which the queries can be executed.  
 */  
Connection getConnection();  
  
/**  
 * called once when the system is shutting down, should be idempotent.  
 */  
void cleanup();  
  
}'''
```

即插即用， 我们支持'org.apache.storm.jdbc.common.HikariCPConnectionProvider'， 这是一个使用HikariCP的实现。

### JdbcMapper

使用JDBC在表中插入数据的主要API是`org.apache.storm.jdbc.mapper.JdbcMapper` 接口：

```
public interface JdbcMapper extends Serializable {  
    List<Column> getColumns(ITuple tuple);  
}
```

`getColumns()` 方法定义了`storm tuple`如何映射到数据库中表示一行的列的列表。

返回的列表的顺序很重要。查询中的占位符以与返回列表相同的顺序进行解析。

例如，如果用户的插入查询是

`insert into user(user_id, user_name, create_date) values (?, ?, now())`，`getColumns`方法返回列表中的第一项将映射到第一位，第二位到第二位，依此类推。我们不会解析提供的查询，以列名称来尝试和解析占位符。没有对查询语法进行任何假设，允许这个连接器被一些非标准的sql框架（如仅支持upsert的Phoenix）使用。

## JdbcInsertBolt

使用 `JdbcInsertBolt`，你可以通过指定一个 `ConnectionProvider` 实例和将 `storm tuple` 转换为DB行的 `JdbcMapper` 实例来构造一个 `JdbcInsertBolt` 实例。另外，您必须使用 `withTableName` 方法提供表名或使用 `withInsertQuery` 插入查询。如果您指定了一个插入查询，那么您应该确保您的 `JdbcMapper` 实例将按照插入查询中的顺序返回一列列表。您可以选择指定一个查询超时秒参数，指定插入查询可以执行的最大秒数。默认设置为 `topology.message.timeout.secs` 的值为 -1 将表示不设置任何查询超时。您应该将查询超时值设置为 `<= topology.message.timeout.secs`。

```
Map hikariConfigMap = Maps.newHashMap();
hikariConfigMap.put("dataSourceClassName", "com.mysql.jdbc.jdbc2.optional.MysqlDataSource");
hikariConfigMap.put("dataSource.url", "jdbc:mysql://localhost/test");
hikariConfigMap.put("dataSource.user", "root");
hikariConfigMap.put("dataSource.password", "password");
ConnectionProvider connectionProvider = new HikariCPConnectionProvider(hikariConfigMap);

String tableName = "user_details";
JdbcMapper simpleJdbcMapper = new SimpleJdbcMapper(tableName, connectionProvider);

JdbcInsertBolt userPersistanceBolt = new JdbcInsertBolt(connectionProvider, simpleJdbcMapper)
    .withTableName("user")
    .withQueryTimeoutSecs(30);
Or
JdbcInsertBolt userPersistanceBolt = new JdbcInsertBolt(connectionProvider, simpleJdbcMapper)
    .withInsertQuery("insert into user values (?,?)")
    .withQueryTimeoutSecs(30);
```

## SimpleJdbcMapper

`storm-jdbc` 包括一个通用的 `JdbcMapper` 实现，称为 `SimpleJdbcMapper`，可以映射 `Storm` 元组到数据库行。`SimpleJdbcMapper` 假定 `storm tuple` 中有与列名相同名称的字段您要写入的数据库表。

要使用 `SimpleJdbcMapper`，你只需要告诉你要写入的 `tableName` 并提供一个 `connectionProvider`

实例。

以下代码创建一个 `SimpleJdbcMapper` 实例：

1. 允许映射器将 `storm tuple` 转换为映射到表 `test.user_details` 中的行的列的列表。 2. 将使用提供的 HikariCP 配置来建立具有指定数据库配置的连接池自动找出您要写入的表的列名称和相应的数据类型。请参阅<https://github.com/brettwooldridge/HikariCP#configuration-knobs-baby> 了解有关 hikari 配置属性的更多信息。

```
Map hikariConfigMap = Maps.newHashMap();
hikariConfigMap.put("dataSourceClassName", "com.mysql.jdbc.jdbc2.optional.MysqlDataSource");
hikariConfigMap.put("dataSource.url", "jdbc:mysql://localhost/test");
hikariConfigMap.put("dataSource.user", "root");
hikariConfigMap.put("dataSource.password", "password");
ConnectionProvider connectionProvider = new HikariCPConnectionProvider(hikariConfigMap);
String tableName = "user_details";
JdbcMapper simpleJdbcMapper = new SimpleJdbcMapper(tableName, connectionProvider);
```

在上面的示例中初始化的映射器假定 `storm tuple` 具有要插入数据的表的所有列的值，其 `getColumns` 方法将按照 Jdbc 连接实例的 `connection.getMetaData().getColumns()` 的顺序返回列。

如果您为 `JdbcInsertBolt` 指定了自己的插入查询，则必须使用显式列显示方式初始化 `SimpleJdbcMapper`，以使模式具有与插入查询相同顺序的列。

例如，如果您的插入查询是 `Insert into user (user_id, user_name) values (?,?)`，那么您的 `SimpleJdbcMapper` 应该使用以下语句进行初始化：

```
java List<Column> columnSchema = Lists.newArrayList( new Column("user_id", java.sql.Types.INTEGER), ... )
```

如果您的 `storm tuple` 仅具有子集列的字段 i.e. 如果表中的某些列具有默认值，并且您只想为没有默认值的列插入值，则可以通过显示的指定 `columnschema` 初始化 `SimpleJdbcMapper`。例如，如果你有一个 `user_details`

表 `create table if not exists user_details (user_id integer, user_name varchar(100), dept_name varchar(100), create_time timestamp)`。要确保只插入没有默认值的列你可以初始化 `jdbcMapper` 如下：

```
List<Column> columnSchema = Lists.newArrayList(
    new Column("user_id", java.sql.Types.INTEGER),
    new Column("user_name", java.sql.Types.VARCHAR),
    new Column("dept_name", java.sql.Types.VARCHAR));
JdbcMapper simpleJdbcMapper = new SimpleJdbcMapper(columnSchema);
```

**JdbcTridentState**

我们还支持持久化trident state 通过使用trident topologies。要创建一个jdbc 持久化的 tridentstate，您需要使用表名或插入查询、JdbcMapper实例和连接提供程序实例进行初始化。见下面的例子：

```
JdbcState.Options options = new JdbcState.Options()
    .withConnectionProvider(connectionProvider)
    .withMapper(jdbcMapper)
    .withTableName("user_details")
    .withQueryTimeoutSecs(30);
JdbcStateFactory jdbcStateFactory = new JdbcStateFactory(options);
```

类似于 `JdbcInsertBolt`，你可以使用 `withInsertQuery` 来指定一个自定义的插入查询，而不是指定一个表名。

## Lookup from Database 查询数据库

我们支持数据库的 `select` 查询以丰富拓扑中的storm tuples。使用 JDBC 执行数据库查询的主要 API 是 `org.apache.storm.jdbc.mapper.JdbcLookupMapper` 接口：

```
void declareOutputFields(OutputFieldsDeclarer declarer);
List<Column> getColumns(ITuple tuple);
List<Values> toTuple(ITuple input, List<Column> columns);
```

`declareOutputFields` 方法用于指明将作为处理 storm tuple 的输出 tuple 的一部分发出哪些字段。

`getColumns` 方法指定选择查询中的占位符列及其 SQL 类型和要使用的值。例如在上面提到的 `user_details` 表中，如果你正在执行一个查询`'select user_name from user_details where user_id = ? and create_time > ?'` `getColumns` 方法将需要一个 storm 输入 tuple，并返回一个包含两个项目的列表。`Column` 类型的 `getValue()` 方法的第一个实例将被用作 `user_id` 的值进行查找，`Column` 类型的 `getValue()` 方法的第二个实例将被用作 `create_time` 的值。注意：返回的列表中的顺序决定了占位符的价值。换句话说，列表中的第一个项目映射在 `select` 查询中第一个'?'，第二个项目是第二个'?'，依次类推。

`toTuple` 方法将 `select` 查询的结果接收输入 tuple 和表示 DB 行的列的列表，并返回要发射的值的列表。

请注意，它返回一个 `Values` 列表，而不仅仅是一个 `Values` 的实例。这允许将单个 DB 行映射到多个输出 storm tuples。

## SimpleJdbcLookupMapper

`storm-jdbc` 包括一个通用的 `JdbcLookupMapper` 实现，叫做 `SimpleJdbcLookupMapper`。

要使用 `SimpleJdbcMapper`，必须使用您的 `bolt` 输出的字段和查询语句中占位符的列的列表来初始化它。以下示例展示如何初始化一个 `SimpleJdbcLookupMapper`，它将

`user_id, user_name, create_date` 声明为输出字段，`user_id` 作为查询语句中的占位符列。`SimpleJdbcMapper` 假定您的 `tuple` 中的字段名称等于占位符列名称，即在我们的示例中，`SimpleJdbcMapper` 将在输入 `tuple` 中查找一个字段 `use_id`，并将其值用作查询语句中占位符的值。对于构造输出 `tuple`，它首先在输入元组中查找 `outputFields` 中指定的字段，如果在输入元组中找不到，那么它会查看 `select query` 的输出行中与列名称相同的列。所以在下面的例子中，如果输入 `tuple` 有字段 `user_id, create_date`，查询语句

是 `select user_name from user_details where user_id = ?`，对于每个输入 `tuple`

`SimpleJdbcLookupMapper.getColumns(tuple)` 将返回 `tuple.getValueByField("user_id")` 用作 `select` 查询的 `?` 中的值。对于 DB 的每个输出行，`SimpleJdbcLookupMapper.toTuple()` 将使用输入元组中的 `user_id, create_date`，因为从结果行只添加 `user_name`，并将这 3 个字段作为单个输出元组返回。

```
Fields outputFields = new Fields("user_id", "user_name", "create_date");
List<Column> queryParamColumns = Lists.newArrayList(new Column("user_id", Types.INTEGER));
this.jdbcLookupMapper = new SimpleJdbcLookupMapper(outputFields, queryParamColumns);
```

## JdbcLookupBolt

要使用 `JdbcLookupBolt`，使用 `ConnectionProvider` 实例，`JdbcLookupMapper` 实例和 `select` 查询来构造一个它的实例。你可以选择指定一个查询超时秒参数，指定 `select` 查询可以采用的最大秒数。默认值为 `topology.message.timeout.secs` 的值。你应该将此值设置为 `<= topology.message.timeout.secs`。

```
String selectSql = "select user_name from user_details where user_id = ?";
SimpleJdbcLookupMapper lookupMapper = new SimpleJdbcLookupMapper(outputFields, queryParamColumns);
JdbcLookupBolt userNameLookupBolt = new JdbcLookupBolt(connectionProvider, selectSql, lookupMapper)
    .withQueryTimeoutSecs(30);
```

## JdbcTridentState for lookup

我们还可以在 `trident topologies` 中查询 `trident state`。

```
JdbcState.Options options = new JdbcState.Options()
    .withConnectionProvider(connectionProvider)
    .withJdbcLookupMapper(new SimpleJdbcLookupMapper(new Fields("user_name"), Lists.newArrayList(
        .withSelectQuery("select user_name from user_details where user_id = ?"))));
```

```
.withQueryTimeoutSecs(30);
```

## Example:

可以在 `src/test/java/topology` 目录中找到一个可运行的例子。

## Setup

- 确保您为您选择的数据库包含JDBC实现依赖关系，作为构建配置的一部分。
- 测试拓扑执行以下查询，因此您的预期数据库必须支持这些查询才能使测试拓扑工作。

```
create table if not exists user (user_id integer, user_name varchar(100), dept_name varchar(100), c  
create table if not exists department (dept_id integer, dept_name varchar(100));  
create table if not exists user_department (user_id integer, dept_id integer);  
insert into department values (1, 'R&D');  
insert into department values (2, 'Finance');  
insert into department values (3, 'HR');  
insert into department values (4, 'Sales');  
insert into user_department values (1, 1);  
insert into user_department values (2, 2);  
insert into user_department values (3, 3);  
insert into user_department values (4, 4);  
select dept_name from department, user_department where department.dept_id = user_department.dept_i
```

## Execution

使用`storm jar`命令运行`org.apache.storm.jdbc.topology.UserPersistanceTopology`类。 参考该课程第5小节`storm jar org.apache.storm.jdbc.topology.UserPersistanceTopology [拓扑名称]`

要使其与Mysql一起工作，您可以将以下内容添加到`pom.xml`中

```
<dependency>  
    <groupId>mysql</groupId>  
    <artifactId>mysql-connector-java</artifactId>  
    <version>5.1.31</version>  
</dependency>
```

您可以使用maven程序集插件生成具有依赖关系的单个jar。 要使用插件，将以下内容添加到您的`pom.xml`并执行 `mvn clean compile assembly:single`

```
<plugin>  
    <artifactId>maven-assembly-plugin</artifactId>  
    <configuration>  
        <archive>  
            <manifest>  
                <mainClass>fully.qualified.MainClass</mainClass>
```

```
</manifest>
</archive>
<descriptorRefs>
    <descriptorRef>jar-with-dependencies</descriptorRef>
</descriptorRefs>
</configuration>
</plugin>
```

## Mysql Example:

```
storm jar ~/repo/incubator-storm/external/storm-jdbc/target/storm-jdbc-0.10.0-SNAPSHOT-jar-with-depe
```

您可以针对应显示新插入的行的用户表执行**select**查询：

```
select * from user;
```

For trident you can view [org.apache.storm.jdbc.topology.UserPersistanceTridentTopology](#).

对于trident，您可以查看[org.apache.storm.jdbc.topology.UserPersistanceTridentTopology](#)。

# Storm JMS 集成

## 关于 Storm JMS

Storm JMS是在Storm框架内集成JMS消息传递的通过框架。

Storm-JMS 允许您通过JMS spout(喷口)将数据注入到Storm， 并通过通用JMS bolt(螺栓)从Storm 消费数据。

JMS Spout(喷口)和Bolt(螺栓)都是数据不可知的。要使用它们， 您需要提供一个简单的Java类， 用于桥接JMS和Storm API 以及封装和特定域的逻辑。

## 组件

### **JMS Spout(喷口)**

JMS Spout(喷口)组件允许将发布到JMS主题或队列的数据由Storm拓扑消费。 JMS Spout(喷口)连接到JMS目标(主题或队列)， 并根据收到的JMS消息的内容发送给Storm "Tuple"对象。

### **JMS Bolt(螺栓)**

JMS Bolt(螺栓)组件允许将Storm 拓扑中的数据发布到JMS目标（主题或队列）。

JMS Bolt(螺栓)连接到JMS目标， 并根据接收的Storm "Tuple"对象发布JMS消息。

## [Example Topology](#)

## [Using Spring JMS](#)

# Storm Redis 集成

## Storm/Trident 集成 Redis

Storm-redis使用Jedis为Redis客户端。

## 用法

如何使用它？

使用它作为一个maven依赖：

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

## 常用 Bolt

Storm-redis提供了基本的Bolt实现， [RedisLookupBolt](#) and [RedisStoreBolt](#)。

根据名称可以知道其功能， [RedisLookupBolt](#) 使用键从Redis中检索值， 而 [RedisStoreBolt](#) 将键/值存储到Redis。 一个元组将匹配一个键/值对， 您可以将匹配模式定义为“[TupleMapper](#)”。

您还可以从[RedisDataTypeDescription](#)中选择数据类型来使用。请参考

[RedisDataTypeDescription.RedisDataType](#)来查看支持哪些数据类型。在一些数据类型（散列和排序集）中， 它需要额外的键和从元组转换的元素成为元素。

这些接口与 [RedisLookupMapper](#) 和 [RedisStoreMapper](#)组合， 分别适合 [RedisLookupBolt](#) 和 [RedisStoreBolt](#)。

## RedisLookupBolt示例

```
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;
}
}

```

```

JedisPoolConfig poolConfig = new JedisPoolConfig.Builder()
    .setHost(host).setPort(port).build();
RedisLookupMapper lookupMapper = new WordCountRedisLookupMapper();
RedisLookupBolt lookupBolt = new RedisLookupBolt(poolConfig, lookupMapper);

```

## RedisStoreBolt示例

```

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");
    }
}

```

```

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

```

## 非简单的 Bolt

如果您的场景不适合 `RedisStoreBolt` 和 `RedisLookupBolt`, `Storm-redis` 还提供了 `AbstractRedisBolt`, 让您扩展和应用业务逻辑。

```

public static class LookupWordTotalCountBolt extends AbstractRedisBolt {
    private static final Logger LOG = LoggerFactory.getLogger(LookupWordTotalCountBolt.class);
    private static final Random RANDOM = new Random();

    public LookupWordTotalCountBolt(JedisPoolConfig config) {
        super(config);
    }

    public LookupWordTotalCountBolt(JedisClusterConfig config) {
        super(config);
    }

    @Override
    public void execute(Tuple input) {
        JedisCommands jedisCommands = null;
        try {
            jedisCommands = getInstance();
            String wordName = input.getStringByField("word");
            String countStr = jedisCommands.get(wordName);
            if (countStr != null) {
                int count = Integer.parseInt(countStr);
                this.collector.emit(new Values(wordName, count));

                // print lookup result with low probability
                if(RANDOM.nextInt(1000) > 995) {
                    LOG.info("Lookup result - word : " + wordName + " / count : " + count);
                }
            } else {
                // skip
                LOG.warn("Word not found in Redis - word : " + wordName);
            }
        } finally {
            if (jedisCommands != null) {

```

```

        returnInstance(jedisCommands);
    }
    this.collector.ack(input);
}
}

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

```

## Trident State 用法

1. RedisState和RedisMapState，它提供Jedis接口，仅用于单次重新启动。
2. RedisClusterState和RedisClusterMapState，它们提供JedisCluster接口，仅用于redis集群。

RedisState

```

```java
JedisPoolConfig poolConfig = new JedisPoolConfig.Builder()
.setHost(redisHost).setPort(redisPort).build();
RedisStoreMapper storeMapper = new
WordCountStoreMapper(); RedisLookupMapper lookupMapper = new
WordCountLookupMapper(); RedisState.Factory factory = new
RedisState.Factory(poolConfig);

```

```

TridentTopology topology = new TridentTopology();
Stream stream = topology.newStream("spout1", spout);

stream.partitionPersist(factory,
                        fields,
                        new RedisStateUpdater(storeMapper).withExpire(86400000),
                        new Fields());

TridentState state = topology.newStaticState(factory);
stream = stream.stateQuery(state, new Fields("word"),
                            new RedisStateQuerier(lookupMapper),
                            new Fields("columnName", "columnValue"));

```

RedisClusterState

```

```java
Set<InetSocketAddress> nodes = new HashSet<InetSocketAddress>();
for (String hostPort : redisHostPort.split(",")) {
    String[] host_port = hostPort.split(":");
    nodes.add(new InetSocketAddress(host_port[0], Integer.valueOf(host_port[1])));
}
JedisClusterConfig clusterConfig = new JedisClusterConfig.Builder().setNodes(nodes)

```

```
        .build();
RedisStoreMapper storeMapper = new WordCountStoreMapper();
RedisLookupMapper lookupMapper = new WordCountLookupMapper();
RedisClusterState.Factory factory = new RedisClusterState.Factory(clusterConfig);

TridentTopology topology = new TridentTopology();
Stream stream = topology.newStream("spout1", spout);

stream.partitionPersist(factory,
    fields,
    new RedisClusterStateUpdater(storeMapper).withExpire(86400000),
    new Fields());

TridentState state = topology.newStaticState(factory);
stream = stream.stateQuery(state, new Fields("word"),
    new RedisClusterStateQuerier(lookupMapper),
    new Fields("columnName","columnValue"));
```

# Azue Event Hubs 集成

针对 Microsoft Azure Eventhubs 的 Storm spout 和 bolt 实现

## build

```
mvn clean package
```

## 运行 topology 示例

要运行 topology 示例, 您需要去修改 config.properties 文件与 eventhubs 配置. 以下是一个例子:

```
eventhubspout.username = [username: policy name in EventHubs Portal]
eventhubspout.password = [password: shared access key in EventHubs Portal]
eventhubspout.namespace = [namespace]
eventhubspout.entitypath = [entitypath]
eventhubspout.partitions.count = [partitioncount]

# if not provided, will use storm's zookeeper settings
# zookeeper.connectionstring=zookeeper0:2181,zookeeper1:2181,zookeeper2:2181

eventhubspout.checkpoint.interval = 10
eventhub.receiver.credits = 1024
```

然后您可以使用 storm.cmd 来提交 topology 示例:

```
storm jar {jarfile} com.microsoft.eventhubs.samples.EventCount {topologyname} {spoutconffile}
where the {jarfile} should be: eventhubs-storm-spout-{version}-jar-with-dependencies.jar
```

## 运行 EventHubSendClient

我们包括一个简单的 EventHubs send client 用于测试. 您可以像下面这样来运行 client:

```
java -cp .\target\eventhubs-storm-spout-{version}-jar-with-dependencies.jar com.microsoft.eventhubs
[username] [password] [entityPath] [partitionId] [messageSize] [messageCount]
```

如果想要发送消息到所有的 partition (分区), 使用 "-1" 作为 partitionId.

## Windows Azure Eventhubs

```
http://azure.microsoft.com/en-us/services/event-hubs/
```

# Storm Elasticsearch 集成

## Storm Elasticsearch Bolt & Trident State

EdIndexBolt, EsPercolateBolt和Estate允许用户将storm中的数据直接传输到Elasticsearch。详细说明请参考以下内容。

### EsIndexBolt (org.apache.storm.elasticsearch.bolt.EsIndexBolt)

EsIndexBolt将tuples直接流入Elasticsearch索。 Tuples以指定的索引和类型组合进行索引。 用户应确保EsTupleMapper可以从输入元组中提取“source”, “index”, “type”和“id”, “index”和“type”用于识别目标索引和类型。“source”一个JSON格式的文档，将在Elasticsearch中编入索引。

```
EsConfig esConfig = new EsConfig(clusterName, new String[]{"localhost:9300"});
EsTupleMapper tupleMapper = new DefaultEsTupleMapper();
EsIndexBolt indexBolt = new EsIndexBolt(esConfig, tupleMapper);
```

### EsPercolateBolt (org.apache.storm.elasticsearch.bolt.EsPercolateBolt)

EsPercolateBolt将tuples直接流入Elasticsearch。 tuples用于发送渗透请求到指定的索引和类型组合。 用户应该确保EsTupleMapper可以从输入元组中提取“source”, “index”, “type”, “index”和“type”用于识别目标索引和类型，“source”是一个文档在JSON格式的字符串将发送到渗透请求到弹性搜索。

```
EsConfig esConfig = new EsConfig(clusterName, new String[]{"localhost:9300"});
EsTupleMapper tupleMapper = new DefaultEsTupleMapper();
EsPercolateBolt percolateBolt = new EsPercolateBolt(esConfig, tupleMapper);
```

如果存在非空的渗透响应，EsPercolateBolt将会为PercolateResponse中每个Percolate.Match发出具有原始源和Percolate.Match的tuple。

### EsState (org.apache.storm.elasticsearch.trident.EsState)

Elasticsearch Trident state也与EsBolts类似。 它将EsConfig和EsTupleMapper作为参数。

```
EsConfig esConfig = new EsConfig(clusterName, new String[]{"localhost:9300"});
EsTupleMapper tupleMapper = new DefaultEsTupleMapper();

StateFactory factory = new EsStateFactory(esConfig, tupleMapper);
TridentState state = stream.partitionPersist(factory, esFields, new EsUpdater(), new Fields());
```

### EsLookupBolt (org.apache.storm.elasticsearch.bolt.EsLookupBolt)

EsLookupBolt对Elasticsearch执行获取请求。为了做到这一点，需要满足三个依赖。除了通常的EsConfig，还必须提供其他两个依赖关系： ElasticsearchGetRequest用于将传入的元组转换为将针对Elasticsearch执行的GetRequest。 EsLookupResultOutput用于声明输出字段，并将GetResponse转换为由bolt发出的值。

传入的tuple被传递给提供的GetRequest创建者，该执行的结果被传递给Elasticsearch客户端。然后， bolt使用提供程序输出适配器（EsLookupResultOutput）将GetResponse转换为值以发送。输出字段也由bolt的用户通过输出适配器（EsLookupResultOutput）指定。

```
EsConfig esConfig = createEsConfig();
ElasticsearchGetRequest getRequestAdapter = createElasticsearchGetRequest();
EsLookupResultOutput output = createOutput();
EsLookupBolt lookupBolt = new EsLookupBolt(esConfig, getRequestAdapter, output);
```

## EsConfig (org.apache.storm.elasticsearch.common.EsConfig)

=提供的组件（Bolt, State）以EsConfig作为构造函数arg。

```
EsConfig esConfig = new EsConfig(clusterName, new String[]{"localhost:9300"});
```

or

```
Map<String, String> additionalParameters = new HashMap<>();
additionalParameters.put("client.transport.sniff", "true");
EsConfig esConfig = new EsConfig(clusterName, new String[]{"localhost:9300"}, additionalParameters)
```

### EsConfig params

Arg	Description	Type
clusterName	Elasticsearch cluster name	String (required)
nodes	Elasticsearch nodes in a String array, each element should follow {host}:{port} pattern	String array (required)
additionalParameters	Additional Elasticsearch Transport Client configuration parameters	Map (optional)

## EsTupleMapper (org.apache.storm.elasticsearch.common.EsTupleMapper)

对于存储在Elasticsearch中的tuple或者从Elasticsearch搜索到的tuple，我们需要定义使用哪些字段。 用户需要通过实现`EsTupleMapper`定义你自己的。 Storm-elasticsearch提供了默认的mapper`org.apache.storm.elasticsearch.common.DefaultEsTupleMapper`，它从相同的字段中提取其源，索引，类型，id值。 您可以参考DefaultEsTupleMapper的实现来看看如何实现自己的。

## Committer Sponsors

- Sriharsha Chintalapani ([@harshach](#))
- Jungtaek Lim ([@HeartSaViR](#))

# Storm MQTT(Message Queuing Telemetry Transport, 消息队列遥测传输) 集成

## About

MQTT是IoT(Internet of Things)应用程序中经常使用的轻量级发布/订阅协议。

Further information can be found at <http://mqtt.org>. The HiveMQ website has a great series on [MQTT Essentials](#).

功能点如下：

- 完整的MQTT支持 (e.g. last will, QoS 0-2, retain, etc.)
- Spout(喷口) 实现订阅MQTT主题
- 用于发布MQTT消息的bolt(螺栓)实现
- 用于发布MQTT消息的trident(三叉线)功能实现
- 身份验证和TLS/SSL支持
- 用户定义的"映射器"用于将MQTT消息转换为元组(订阅者)
- 用户定义的"映射器" User-defined "mappers" 用于将元组转换为MQTT消息(发布者)

## 快速开始

要快速查看MQTT集成操作，请按照以下说明进行操作。

### 启动 MQTT 的代理和发布者

以下命令将在端口1883上创建一个MQTT代码，并启动一个随机发布的发布者温度/湿度值到MQTT主题。

打开终端并执行以下命令(根据需要更改路径):

```
java -cp examples/target/storm-mqtt-examples-*-SNAPSHOT.jar org.apache.storm.mqtt.examples.MqttBrok
```

### 运行示例拓扑

使用Flux运行示例拓扑。这将启动由MQTT Spout(喷口)组成的本地模式集群和拓扑发布到只收录信息的 bolt(螺栓)。

在单独的终端中，运行以下命令(请注意，“storm”可执行文件必须位于你的PATH上):

```
storm jar ./examples/target/storm-mqtt-examples-* -SNAPSHOT.jar org.apache.storm.flux.Flux ./example
```

您应该可以看到来自MQTT的数据被bolt(螺栓)记录：

```
27020 [Thread-17-log-executor[3 3]] INFO o.a.s.f.w.b.LogInfoBolt - {user=tgoetz, deviceId=1234, lo  
27030 [Thread-17-log-executor[3 3]] INFO o.a.s.f.w.b.LogInfoBolt - {user=tgoetz, deviceId=1234, lo  
27040 [Thread-17-log-executor[3 3]] INFO o.a.s.f.w.b.LogInfoBolt - {user=tgoetz, deviceId=1234, lo  
27049 [Thread-17-log-executor[3 3]] INFO o.a.s.f.w.b.LogInfoBolt - {user=tgoetz, deviceId=1234, lo  
27059 [Thread-17-log-executor[3 3]] INFO o.a.s.f.w.b.LogInfoBolt - {user=tgoetz, deviceId=1234, lo  
27069 [Thread-17-log-executor[3 3]] INFO o.a.s.f.w.b.LogInfoBolt - {user=tgoetz, deviceId=1234, lo
```

允许本地集群退出，或者通过键入Ctrl-C 来停止。

## MQTT容错实战

在拓扑关闭之后，MQTT Spout(喷口)创建的MQTT订阅将与代理持续存在，并且它将继续接收和排队消息(只要代理正在运行)。

如果您再次运行拓扑(当代理程序仍在运行时)，当spout(喷口)最初连接到MQTT代理时，它会收到所有的信息，错过的信息就当作失败的消息。您应该看到这是一个消息的爆发，接着再以每秒两个消息的速率显示。

这是因为在默认情况下，MQTT Spout(喷口)在订阅时会创建一个会话这意味着它要求它的代理在离线时持有并重新提交其错过的任何消息。另外一个重要因素是'MqttBrokerPublisher'发布MQTT Qos 为'1'的消息,这意味着至少一次交付。

有关MQTT容错的更多信息，请参阅下文的交付保证部分。

## 交付保证

在Storm中，**MQTT Spout**(喷口)至少提供一次传递，具体取决于发布者的配置以及MQTT的spout(喷口)。

MQTT协议定义了以下QoS级别：

- 0 - 最多一次 (AKA "Fire and Forget")
- 1 - 起码一次
- 2 - 完全一次

这里可能有点混乱，因为MQTT协议规范并没有真正解决一个节点补一个灾难性事件完全焚

烧的事实。这与Storm的可靠性形成了一个鲜明的对比，该模型期望并拥抱节点的概念。

所以弹性最终取决于基础的MQTT实现和基础架构。

## 推荐

\*你将永远不会得到一次处理这个spout(喷口)。它可以与三叉线一起使用，但不会提供事物语义。

如果您需要可靠性的保证(即 至少一次处理):

1. 对于MQTT发布者(Storm之外),发布QoS为'1'的消息，以便在spout脱机时，代理保存消息。
2. 使用spout的默认值 (`cleanSession = false and qos = 1`)
3. 如果可以，请确保接收和MQTT消息和任何结果是幂等的。
4. 确保您的MQTT代理不会因为网络分区而死亡或孤立。为自然灾害和人为灾害及网络分区做好准备。以及焚化和破坏的发生。

## 配置

有关完整的配置选项，请参阅JavaDoc for `org.apache.storm.mqtt.common.MqttOptions`.

## 信息映射器

要定义MQTT消息如何映射到Storm元组，您可以使用该实现配置MQTT spout `org.apache.storm.mqtt.MqttMessageMapper` 接口，如下所示:

```
public interface MqttMessageMapper extends Serializable {  
    Values toValues(MqttMessage message);  
    Fields outputFields();  
}
```

`MqttMessage` 类包含消息发布的主题("String")和消息的有效负载(`byte[]`)。例如，这是一个 `MqttMessageMapper` 实现，它基于这两者的内容生成元组消息主题和有效负载:

```
/**  
 * Given a topic name: "users/{user}/{location}/{deviceId}"  
 * and a payload of "{temperature}/{humidity}"  
 * emits a tuple containing user(String), deviceId(String), location(String), temperature(float), h  
 */  
public class CustomMessageMapper implements MqttMessageMapper {
```

```
private static final Logger LOG = LoggerFactory.getLogger(CustomMessageMapper.class);

public Values toValues(MqttMessage message) {
    String topic = message.getTopic();
    String[] topicElements = topic.split("/");
    String[] payloadElements = new String(message.getMessage()).split("/");

    return new Values(topicElements[2], topicElements[4], topicElements[3], Float.parseFloat(pa
        Float.parseFloat(payloadElements[1])));
}

public Fields outputFields() {
    return new Fields("user", "deviceId", "location", "temperature", "humidity");
}
}
```

## 元组映射器

使用MQTT bolt(螺栓)或Trident功能发布MQTT消息时，您需要将元组数据映射到MQTT消息(主题/负载)。这是通过实现 `org.apache.storm.mqtt.MqttTupleMapper` 接口完成的：

```
public interface MqttTupleMapper extends Serializable{

    MqttMessage toMessage(ITuple tuple);

}
```

例如，一个简单的 `MqttTupleMapper` 实现可能如下所示：

```
public class MyTupleMapper implements MqttTupleMapper {
    public MqttMessage toMessage(ITuple tuple) {
        String topic = "users/" + tuple.getStringByField("userId") + "/" + tuple.getStringByField("user
        byte[] payload = tuple.getStringByField("message").getBytes();
        return new MqttMessage(topic, payload);
    }
}
```

## MQTT Spout(喷口)平行度

建议您对MQTT spout(喷口)使用并行度1，否则最终会出现多个实例的端口订阅机同的主题，导致重复消费。

如果您要并行化spout(喷口)，建议您在拓扑中使用多个spout(喷口)实例，并使用MQTT主题选择器对数据进行分组。如何实现分区的策略的方法最终由您的MQTT主题结构决定。举个例子，如果你按区域划分主题(如东/西)，你可以做类似于以下内容：

```
String spout1Topic = "users/east/#";
String spout2Topic = "users/west/#";
```

然后通过预订一个(bolt)螺栓将每个流加入到结果流中。

## 使用 Flux

以上Flux YAML 配置创建了示例中使用的拓扑：

```
name: "mqtt-topology"

components:
  ##### MQTT Spout Config #####
  - id: "mqtt-type"
    className: "org.apache.storm.mqtt.examples.CustomMessageMapper"

  - id: "mqtt-options"
    className: "org.apache.storm.mqtt.common.MqttOptions"
    properties:
      - name: "url"
        value: "tcp://localhost:1883"
      - name: "topics"
        value:
          - "/users/tgoetz/#"

# topology configuration
config:
  topology.workers: 1
  topology.max.spout.pending: 1000

# spout definitions
spouts:
  - id: "mqtt-spout"
    className: "org.apache.storm.mqtt.spout.MqttSpout"
    constructorArgs:
      - ref: "mqtt-type"
      - ref: "mqtt-options"
    parallelism: 1

# bolt definitions
bolts:
  - id: "log"
    className: "org.apache.storm.flux.wrappers.bolts.LogInfoBolt"
    parallelism: 1

streams:
  - from: "mqtt-spout"
    to: "log"
    grouping:
      type: SHUFFLE
```

## 使用 Java

同样，你可以使用Storm Core Java API 来创建相同的拓扑结构：

```
TopologyBuilder builder = new TopologyBuilder();
MqttOptions options = new MqttOptions();
options.setTopics(Arrays.asList("/users/tgoetz/#"));
options.setCleanConnection(false);
MqttSpout spout = new MqttSpout(new StringMessageMapper(), options);

MqttBolt bolt = new LogInfoBolt();

builder.setSpout("mqtt-spout", spout);
builder.setBolt("log-bolt", bolt).shuffleGrouping("mqtt-spout");

return builder.createTopology();
```

## SSL/TLS

如果要连接的MQTT代理需要SSL或SSL客户端身份验证，则需要配置具有适当URI的端口以及包含必要证书的密钥库/信任库文件的位置。

### SSL/TLS URLs

要通过SSL/TLS连接使用前缀为 `ssl://` 或 `tls://` 而不是 `tcp://`. 进一步控制该算法可以指定一个特定的协议：

- `ssl://` 使用JVM默认版本的SSL 协议。
- `sslv*://` 使用特定版本的SSL协议，其中 \* 替换为版本 (e.g. `sslv3://`)
- `tls://` 使用JVM默认版本的TLS 协议。
- `tlsv*://` 使用特定版本的TLS 协议，其中 \* 替换为版本 (e.g. `tlsv1.1://`)

指定 密钥库/信任域的位置

`MqttSpout`, `MqttBolt` and `MqttPublishFunction` 都有构造函数，它们使用一个 `KeyStoreLoader` 实例来加载 TLS/SSL连接所需的证书。例如：

```
public MqttSpout(MqttMessageMapper type, MqttOptions options, KeyStoreLoader keyStoreLoader)
```

`DefaultKeyStoreLoader` 类可用于从本地文件系统加载证书。请注意，密钥库/信任库需要在可能执行`spout`(喷口)/`bolt`(螺栓)的所有工作节点上可用。要使用 `DefaultKeyStoreLoader` 指定密钥库/信任库文件的位置，并设置必要的密码：

```
DefaultKeyStoreLoader ksl = new DefaultKeyStoreLoader("/path/to/keystore.jks", "/path/to/truststore")
```

```
ksl.setKeyStorePassword("password");
ksl.setTrustStorePassword("password");
//...
```

III

如果您的密钥库/信任库证书存储在单个文件中，则可以使用单参数构架函数：

```
DefaultKeyStoreLoader ksl = new DefaultKeyStoreLoader("/path/to/keystore.jks");
ksl.setKeyStorePassword("password");
//...
```

还可以使用Flux来配置SSL/TLS：

```
name: "mqtt-topology"

components:
    ##### MQTT Spout Config #####
    - id: "mqtt-type"
        className: "org.apache.storm.mqtt.examples.CustomMessageMapper"

    - id: "keystore-loader"
        className: "org.apache.storm.mqtt.ssl.DefaultKeyStoreLoader"
        constructorArgs:
            - "keystore.jks"
            - "truststore.jks"
    properties:
        - name: "keyPassword"
            value: "password"
        - name: "keyStorePassword"
            value: "password"
        - name: "trustStorePassword"
            value: "password"

    - id: "mqtt-options"
        className: "org.apache.storm.mqtt.common.MqttOptions"
        properties:
            - name: "url"
                value: "ssl://raspberrypi.local:8883"
            - name: "topics"
                value:
                    - "/users/tgoetz/#"

# topology configuration
config:
    topology.workers: 1
    topology.max.spout.pending: 1000

# spout definitions
spouts:
    - id: "mqtt-spout"
        className: "org.apache.storm.mqtt.spout.MqttSpout"
        constructorArgs:
            - ref: "mqtt-type"
            - ref: "mqtt-options"
            - ref: "keystore-loader"
```

```
parallelism: 1

# bolt definitions
bolts:

- id: "log"
  className: "org.apache.storm.flux.wrappers.bolts.LogInfoBolt"
  parallelism: 1

streams:

- from: "mqtt-spout"
  to: "log"
  grouping:
    type: SHUFFLE
```

## Committer Sponsors

- P. Taylor Goetz ([ptgoetz@apache.org](mailto:ptgoetz@apache.org))

# Storm MongoDB 集成

Storm/Trident集成MongoDB。该包中包括核心bolts和trident states，允许storm topology将 storm tuples插入到数据库集合中，或者针对storm topology中的数据库集合执行更新查询。

## Insert into Database

此包中包含用于将数据插入数据库集合的bolt和trident state。

### MongoMapper

使用MongoDB在集合中插入数据的主要API是

`org.apache.storm.mongodb.common.mapper.MongoMapper` 接口：

```
public interface MongoMapper extends Serializable {  
    Document toDocument(ITuple tuple);  
}
```

### SimpleMongoMapper

storm-mongodb包括一个通用的MongoMapper实现，称为SimpleMongoMapper，可以将Storm元组映射到一个数据库文件。SimpleMongoMapper假定storm tuple具有与您要写入的数据库集合中的文档字段名称相同的字段。

```
public class SimpleMongoMapper implements MongoMapper {  
    private String[] fields;  
  
    @Override  
    public Document toDocument(ITuple tuple) {  
        Document document = new Document();  
        for(String field : fields){  
            document.append(field, tuple.getValueByField(field));  
        }  
        return document;  
    }  
  
    public SimpleMongoMapper withFields(String... fields) {  
        this.fields = fields;  
        return this;  
    }  
}
```

### MongoInsertBolt

要使用MongoInsertBolt，您可以通过指定url，collectionName和将 storm tuple转换为DB文档的MongoMapper实现来构造它的一个实例。以下是标准的URI连接方案：

```
mongodb://[username:password@]host1[:port1][,host2[:port2],...,hostN[:portN]]]/[database][?options]
```

有关Mongo URI的更多选项信息（例如：写关注选项），您可以访问

<https://docs.mongodb.org/manual/reference/connection-string/#connections-connection-options>

```
String url = "mongodb://127.0.0.1:27017/test";
String collectionName = "wordcount";

MongoMapper mapper = new SimpleMongoMapper()
    .withFields("word", "count");

MongoInsertBolt insertBolt = new MongoInsertBolt(url, collectionName, mapper);
```

## MongoTridentState

我们还支持在trident topologies中持久化trident state。要创建一个Mongo持久的trident state，您需要使用url, collectionName, “MongoMapper”实例初始化它。见下面的例子：

```
MongoMapper mapper = new SimpleMongoMapper()
    .withFields("word", "count");

MongoState.Options options = new MongoState.Options()
    .withUrl(url)
    .withCollectionName(collectionName)
    .withMapper(mapper);

StateFactory factory = new MongoStateFactory(options);

TridentTopology topology = new TridentTopology();
Stream stream = topology.newStream("spout1", spout);

stream.partitionPersist(factory, fields, new MongoStateUpdater(), new Fields());
```

## NOTE:

如果没有提供唯一的索引，在发生故障的情况下，trident state插入可能会导致重复的文档。

## Update from Database

包中包含用于从数据库集合更新数据的bolt。

## SimpleMongoUpdateMapper

storm-mongodb包括一个通用的MongoMapper实现，称为SimpleMongoUpdateMapper，可以将Storm元组映射到数据库文档。SimpleMongoUpdateMapper假定风暴元组具有与您要写入的数据库集合中的

文档字段名称相同的字段。`SimpleMongoUpdateMapper`使用`$ set`运算符来设置文档中字段的值。有关更新操作的更多信息，可以访问

<https://docs.mongodb.org/manual/reference/operator/update/>

```
public class SimpleMongoUpdateMapper implements MongoMapper {  
    private String[] fields;  
  
    @Override  
    public Document toDocument(ITuple tuple) {  
        Document document = new Document();  
        for(String field : fields){  
            document.append(field, tuple.getValueByField(field));  
        }  
        return new Document("$set", document);  
    }  
  
    public SimpleMongoUpdateMapper withFields(String... fields) {  
        this.fields = fields;  
        return this;  
    }  
}
```

## QueryFilterCreator

用于创建MongoDB查询过滤器的主要API是`org.apache.storm.mongodb.common.QueryFilterCreator`接口：

```
public interface QueryFilterCreator extends Serializable {  
    Bson createFilter(ITuple tuple);  
}
```

## SimpleQueryFilterCreator

`storm-mongodb`包括一个通用的`QueryFilterCreator`实现，称为`simpleQueryFilterCreator`，可以通过给定的`Tuple`创建一个MongoDB查询过滤器。`QueryFilterCreator`使用`$ eq`运算符匹配等于指定值的值。有关查询运算符的更多信息，可以访问

<https://docs.mongodb.org/manual/reference/operator/query/>

```
public class SimpleQueryFilterCreator implements QueryFilterCreator {  
    private String field;  
  
    @Override  
    public Bson createFilter(ITuple tuple) {  
        return Filters.eq(field, tuple.getValueByField(field));  
    }  
  
    public SimpleQueryFilterCreator withField(String field) {  
        this.field = field;  
    }
```

```
    return this;
}

}
```

## MongoUpdateBolt

要使用 `MongoUpdateBolt`, 你可以通过指定 `Mongo url`, `collectionName`, 一个 `QueryFilterCreator` 实现和一个 ```MongoMapper` 实现来将 `storm tuple` 转换成 `DB` 文档来构造一个实例。

```
MongoMapper mapper = new SimpleMongoUpdateMapper()
    .withFields("word", "count");

QueryFilterCreator updateQueryCreator = new SimpleQueryFilterCreator()
    .withField("word");

MongoUpdateBolt updateBolt = new MongoUpdateBolt(url, collectionName, updateQueryCreator, m

//if a new document should be inserted if there are no matches to the query filter
//updateBolt.withUpsert(true);
```

或者为 `QueryFilterCreator` 使用匿名内部类实现:

```
MongoMapper mapper = new SimpleMongoUpdateMapper()
    .withFields("word", "count");

QueryFilterCreator updateQueryCreator = new QueryFilterCreator() {
    @Override
    public Bson createFilter(ITuple tuple) {
        return Filters.gt("count", 3);
    }
};

MongoUpdateBolt updateBolt = new MongoUpdateBolt(url, collectionName, updateQueryCreator, m

//if a new document should be inserted if there are no matches to the query filter
//updateBolt.withUpsert(true);
```

# Storm OpenTSDB 集成

## Storm OpenTSDB Bolt 和 TridentState

OpenTSDB 为时间序列数据提供了可扩展且高可用性的存储. 它由 Time Series Daemon (TSD) servers 以及命令行工具组成. 每个 TSD 连接到配置的 HBase 集群以 push/query (推送/查询) 数据.

时间序列数据点包括: - a metric name. - a UNIX timestamp (seconds or milliseconds since Epoch). - a value (64 bit integer or single-precision floating point value). - a set of tags (key-value pairs) that describe the time series the point belongs to.

Storm bolt 和 trident state 从一个基于给定的 `TupleMetricPointMapper` 的 tuple 中创建了上面的时间序列数据.

该模块提供了 core Storm 和 Trident bolt 实现, 用户将数据写入 OpenTSDB.

时间序列数据点被写入时有 **at-least-once** (至少一次) 的语义保证, 并且重复的数据点应该像 OpenTSDB 的 [这里](#) 提及的一样来处理.

## 示例

### Core Bolt

下面的示例描述了 `org.apache.storm.opentsdb.bolt.OpenTsdbBolt` cord bolt 的使用方法

```
OpenTsdbClient.Builder builder = OpenTsdbClient.newBuilder(openTsdbUrl).sync(30_000).returnValue();
final OpenTsdbBolt openTsdbBolt = new OpenTsdbBolt(builder, Collections.singletonList(TupleMetricPointMapper.DEFAULT_MAPPER));
openTsdbBolt.withBatchSize(10).withFlushInterval(2000);
topologyBuilder.setBolt("opentsdb", openTsdbBolt).shuffleGrouping("metric-gen");
```

### Trident State

```
final OpenTsdbStateFactory openTsdbStateFactory =
    new OpenTsdbStateFactory(OpenTsdbClient.newBuilder(tsdbUrl),
                            Collections.singletonList(TupleOpenTsdbDatapointMapper.DEFAULT_MAPPER));
TridentTopology tridentTopology = new TridentTopology();

final Stream stream = tridentTopology.newStream("metric-tsdb-stream", new MetricGenSpout());
stream.peek(new Consumer() {
```

```
@Override  
public void accept(TridentTuple input) {  
    LOG.info("##### Received tuple: [{}]", input);  
}  
}).partitionPersist(openTsdbStateFactory, MetricGenSpout.DEFAULT_METRIC_FIELDS, new OpenTsd
```

# Storm Kinesis 集成

## Storm Kinesis Spout

提供的核心storm spout(喷口)，用户从Amzon Kinesis Streams 中的流中消费数据。它存储可以在zookeeper中提交的序列号，并在重新启动后默认启动消息记录。下面是创建使用spout的示例拓扑的代码示例。下面说明配置spout(喷口)时使用的每个对象。理想情况下，spout(喷口)任务的数量应等于运动时间碎片的数量。但是，每个任务都可以从多个分片中读取。

```
public class KinesisSpoutTopology {
    public static void main (String args[]) throws InvalidTopologyException, AuthorizationException
        String topologyName = args[0];
        RecordToTupleMapper recordToTupleMapper = new TestRecordToTupleMapper();
        KinesisConnectionInfo kinesisConnectionInfo = new KinesisConnectionInfo(new CredentialsProv
            1000);
        ZkInfo zkInfo = new ZkInfo("localhost:2181", "/kinesisOffsets", 20000, 15000, 10000L, 3, 20
        KinesisConfig kinesisConfig = new KinesisConfig(args[1], ShardIteratorType.TRIM_HORIZON,
            recordToTupleMapper, new Date(), new ExponentialBackoffRetrier(), zkInfo, kinesisCo
        KinesisSpout kinesisSpout = new KinesisSpout(kinesisConfig);
        TopologyBuilder topologyBuilder = new TopologyBuilder();
        topologyBuilder.setSpout("spout", kinesisSpout, 3);
        topologyBuilder.setBolt("bolt", new KinesisBoltTest(), 1).shuffleGrouping("spout");
        Config topologyConfig = new Config();
        topologyConfig.setDebug(true);
        topologyConfig.setNumWorkers(3);
        StormSubmitter.submitTopology(topologyName, topologyConfig, topologyBuilder.createTopology(
    }
}
```

就像你可以看到的，在它的构造函数中，引出了一个KinesisConfig对象。KinesisConfig的构造函数需要8个对象，如下所描述。

### String streamName

用于消费数据的kinesis时间流的名称

### ShardIteratorType shardIteratorType

支持3种类型 - TRIM\_HORIZON(beginning of shard), LATEST and AT\_TIMESTAMP. 默认情况下，如果分片状态为this，则忽略此参数在zookeeper中寻找。所以它们将首次应用从拓扑开始。如果您想在拓扑的后续运行中使用任何这些拓扑，将需要清除用于存储序列号的zookeeper节点的状态。

## RecordToTupleMapper recordToTupleMapper

一个 `RecordToTupleMapper` 接口的实现，该端口将会将 `kinesis` 记录转换为 `storm` 元组。它有两种方法。`getOutputFields` 告诉 `spout` 要从 `getTuple` 方法发出的元组中存在的数据。如果 `getTuple` 返回 `null`，记录将被确认。

```
java Fields getOutputFields (); List<Object> getTuple (Record record);
```

## Date timestamp

与 `AT_TIMESTAMP sharedIteratorType` 参数结合使用。这将使得 `spout` (喷口) 从那时开始提取记录。该 `kinesis` 时间使用的时间是与 `kinesis` 时间记录相关的服务器端时间。

## FailedMessageRetryHandler failedMessageRetryHandler

`FailedMessageRetryHandler` 接口的实现。默认情况下，该模块提供支持指数退避重试的实现失败消息的机制。该实现有两个构造函数。默认值没有 `args` 构造函数将配置在 100 毫秒的第一次重试随后在 `Math.pow(2,i-1)` 中退出，其中 `i` 是范围 2 中的重试次数到 `LONG.MAX_LONG`。2 表示指数函数的基数(秒)。另外一个构造函数以毫秒为单位进行重试间隔，作为第一个参数进行首次重试，以秒为单位的指数函数为第二个参数，重试作为第三个参数。这种接口的方法及其 `spout` (喷口) 的工作方式如下描述

```
java boolean failed (KinesisMessageId messageId); KinesisMessageId getNextFailedMessageToRetry (); void
```

将在每个发生故障的元组上调用失败的方法。如果计划重新尝试失败的消息，则返回 `true`，否则返回 `false`。

`getNextFailedMessageToRetry` 方法将被称为第一件事，每次一个 `spout` (喷口) 想要发生一个元组。它应该返回一个应该重试的消息，如果有，否则为空。请注意，在该时间段内没有任何重试信息的情况下，它可以返回 `null`。但是，当该消息被调用失败方法时，它最终将返回它返回 `true` 的每个消息。

如果 `spout` (喷口) 成功的设法从 `kinesis` 时获取记录并发送，`failedMessageEmitted` 将被调用。否则，当 `getNextFailedMessageToRetry` 方法被再次调用的时候，该实现应该会返回相同的消息。

一旦失败的消息被重新发送并被 `spout` (喷口) 成功地确认，则会被呼叫。如果失败，`spout` (喷口) 失败就会被再次呼叫。

## ZkInfo zkInfo

封装的对象信息用来zookeeper的交互。这个构造函数使用zkUrl作为第一个参数，它是一个逗号分隔的字符串zk host和端口，zkNode作为第二个参数将用作存储提交序列号的根节点，会话超时以毫秒为单位，连接超时作为第四毫秒，提交间隔为提交序列号到zookeeper的毫秒数的五分之一，重试尝试作为zk客户端的第六个连接重试尝试，重试间隔以毫秒为单位的等待时间，然后再重试连接。

## KinesisConnectionInfo kinesisConnectionInfo

使用kinesis客户端捕获连接到kinesis的参数的对象。它有一个构造函数来实现AWSCredentialsProvider作为第一个参数。该模块提供了一个称为 CredentialsProviderChain 的实现，它允许spout(喷口)使用以下之一进行kinesis检测，这5个机制顺序按照以下顺序 - EnvironmentVariableCredentialsProvider, SystemPropertiesCredentialsProvider, ClasspathPropertiesFileCredentialsProvider, InstanceProfileCredentialsProvider, ProfileCredentialsProvider。它需要一个ClientConfiguration 对象作为配置。它需要一个ClientConfiguration 对象作为配置运动的第二个参数客户端，Regions 作为第三个参数，设置客户端上要连接的区域，recordsLimit作为第四个参数，表示最大数量的记录Kinesis客户端将每个GetReocrds请求检索。这个限制应该根据记录的大小，运动时间来仔细选择吞吐量限制和风暴中每个元组延迟的拓扑。另外如果一个任务将从多个分片读取，那么这将影响选择权限认证。

## Long maxUncommittedRecords

这表示每个任务允许的最大未提交序列号数。一旦达到这个数字，spout就不会从kinesis中获取任何新的记录。未提交的序列号被定义为尚未提交给zookeeper的任务的所有消息的总和。这与拓扑级别最大待处理消息不同。例如，如果此值设置为10，并且spout将序列号从1发送到10。序号1正在等待，2到10次被告知。在这种情况下，未提交的序列的数量为10，因为1到10范围内的序列号可以被提交到zk。但是，storm仍然可以在端口上调用下一个元组，因为只有一个等待消息。

## Maven dependencies

Aws sdk version that this was tested with is 1.10.77

```
<dependencies>
    <dependency>
        <groupId>com.amazonaws</groupId>
        <artifactId>aws-java-sdk</artifactId>
        <version>${aws-java-sdk.version}</version>
```

```
<scope>provided</scope>
</dependency>
<dependency>
    <groupId>org.apache.storm</groupId>
    <artifactId>storm-core</artifactId>
    <version>${project.version}</version>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>org.apache.curator</groupId>
    <artifactId>curator-framework</artifactId>
    <version>${curator.version}</version>
    <exclusions>
        <exclusion>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
        </exclusion>
        <exclusion>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-log4j12</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>com.googlecode.json-simple</groupId>
    <artifactId>json-simple</artifactId>
</dependency>
</dependencies>
```

## 将来的工作

处理kinesis中的碎片的合并或分裂， Trident喷口实施和指标

# Storm Druid 集成

## Storm Druid Bolt 和 TridentState

该模块提供了将数据写入Druid 数据存储的核心Strom和Trident bolt(螺栓)的实现。 该实现使用Druid's的Tranquility库向druid发送消息。

一些实施细节从现有的借用 [Tranquility Storm Bolt](#). 这个新的Bolt(螺栓)增加了支持最新的 storm释放， 并保持在storm回购的bolt(螺栓)。

### Core Bolt

下面的例子描述了使用 `org.apache.storm.druid.bolt.DruidBeamBolt` 的核心bolt(螺栓)默认情况下，该bolt(螺栓)希望收到元组，其中"事件"字段提供您的事件类型。可以通过实现 `ITupleDruidEventMapper` 接口来更改此逻辑。

```
DruidBeamFactory druidBeamFactory = new SampleDruidBeamFactoryImpl(new HashMap<String, Object>())
DruidConfig druidConfig = DruidConfig.newBuilder().discardStreamId(DruidConfig.DEFAULT_DISCARD_S
ITupleDruidEventMapper<Map<String, Object>> eventMapper = new TupleDruidEventMapper<>(TupleDruid
DruidBeamBolt<Map<String, Object>> druidBolt = new DruidBeamBolt<Map<String, Object>>(druidBeamF
topologyBuilder.setBolt("druid-bolt", druidBolt).shuffleGrouping("event-gen");
topologyBuilder.setBolt("printer-bolt", new PrinterBolt()).shuffleGrouping("druid-bolt" , druidC
```

### Trident State

```
DruidBeamFactory druidBeamFactory = new SampleDruidBeamFactoryImpl(new HashMap<String, Object>()
ITupleDruidEventMapper<Map<String, Object>> eventMapper = new TupleDruidEventMapper<>(TupleDrui
final Stream stream = tridentTopology.newStream("batch-event-gen", new SimpleBatchSpout(10));

stream.peek(new Consumer() {
    @Override
    public void accept(TridentTuple input) {
        LOG.info("##### Received tuple: {}", input);
    }
}).partitionPersist(new DruidBeamStateFactory<Map<String, Object>>(druidBeamFactory, eventMapper
```

样品工厂实现

Druid bolt 必须配置一个 BeamFactory. 您可以使用它们其中一个来实现 [DruidBeams builder's "buildBeam\(\)" method](#). See the [Configuration documentation](#) for details. For more

details refer [Tranquility library](#) docs.

```
public class SampleDruidBeamFactoryImpl implements DruidBeamFactory<Map<String, Object>> {

    @Override
    public Beam<Map<String, Object>> makeBeam(Map<?, ?> conf, IMetricsContext metrics) {

        final String indexService = "druid/overlord"; // The druid.service name of the indexing ser
        final String discoveryPath = "/druid/discovery"; // Curator service discovery path. config:
        final String dataSource = "test"; //The name of the ingested datasource. Datasources can be
        final List<String> dimensions = ImmutableList.of("publisher", "advertiser");
        List<AggregatorFactory> aggregators = ImmutableList.<AggregatorFactory>of(
            new CountAggregatorFactory(
                "click"
            )
        );
        // Tranquility needs to be able to extract timestamps from your object type (in this case,
        final Timestamper<Map<String, Object>> timestamper = new Timestamper<Map<String, Object>>()
        {
            @Override
            public DateTime timestamp(Map<String, Object> theMap)
            {
                return new DateTime(theMap.get("timestamp"));
            }
        };
        // Tranquility uses ZooKeeper (through Curator) for coordination.
        final CuratorFramework curator = CuratorFrameworkFactory
            .builder()
            .connectString((String)conf.get("druid.tranquility.zk.connect")) //take config from
            .retryPolicy(new ExponentialBackoffRetry(1000, 20, 30000))
            .build();
        curator.start();

        // The JSON serialization of your object must have a timestamp field in a format that Druid
        // Druid expects the field to be called "timestamp" and to be an ISO8601 timestamp.
        final TimestampSpec timestampSpec = new TimestampSpec("timestamp", "auto", null);

        // Tranquility needs to be able to serialize your object type to JSON for transmission to D
        // done with Jackson. If you want to provide an alternate serializer, you can provide your
        // In this case, we won't provide one, so we're just using Jackson.
        final Beam<Map<String, Object>> beam = DruidBeams
            .builder(timestamper)
            .curator(curator)
            .discoveryPath(discoveryPath)
            .location(DruidLocation.create(indexService, dataSource))
            .timestampSpec(timestampSpec)
            .rollup(DruidRollup.create(DruidDimensions.specific(dimensions), aggregators, Query
            .tuning(
                ClusteredBeamTuning
                    .builder()
                    .segmentGranularity(Granularity.HOUR)
                    .windowPeriod(new Period("PT10M"))
                    .partitions(1)
                    .replicants(1)
            )
        )
    }
}
```

```
        .build()
    )
.druidBeamConfig(
    DruidBeamConfig
        .builder()
        .indexRetryPeriod(new Period("PT10M"))
        .build())
.buildBeam();

return beam;
}
}
```



Example code is available [here](#).

# Storm and Kestrel

本页介绍如何使用Storm消费来自Kestrel集群的项目。

## 准备阶段

### Storm

本教程使用的示例来自于 [storm-kestrel](#) 项目和 [storm-starter](#) 项目。建议你克隆这些项目并跟着示例走。阅读 [Setting up development environment](#) 和 [Creating a new Storm project](#) 来设置你的机器。

### Kestrel

它假设您可以在本地选择Kestrel 服务器，如上所述 [here](#).

### Kestrel 服务 和 队列

单个kestrel服务器具有一组队列。Kestrel 队列是在JVM上运行的非常简单的消息队列，并使用memcache协议(具有一定的扩展名)与客户端进行通信。对于更加详细的信息，你可查看 [KestrelThriftClient](#) 类里面提供的 [storm-kestrel](#) 项目.

每个队列按照FIFO(先进先出)的原则进行严格排序。跟随性能项目缓存在系统内存中；但是，只有前128MB保存在内存中。当服务器停止时，队列状态存储在日志文件中。

此外，还可以从 [here](#) 找到细节。

Kestrel is: \* fast \* small \* durable(耐久) \* reliable(稳定)

例如，Twitter 使用 Kestrel 作为其消息传递基础设施的骨干，如上所述[here](#).

### 添加项目至 Kestrel

首先，我们需要一个可以将项目添加到Kestrel队列的程序。以下方法受益于 KestrelClient 的实现 [storm-kestrel](#). 它将句子添加到从包含五个可能句子的数组中随机选择的Kestrel队列中。

```
private static void queueSentenceItems(KestrelClient kestrelClient, String queueName)
    throws ParseError, IOException {

    String[] sentences = new String[] {
        "the cow jumped over the moon",
        "an apple a day keeps the doctor away",
        "i am a little teapot just six inches tall",
        "i have a tail and two little ears",
        "one in the middle and one on each ear"
    };

    for (String sentence : sentences) {
        kestrelClient.addMessage(queueName, sentence);
    }
}
```

```

    "four score and seven years ago",
    "snow white and the seven dwarfs",
    "i am at two with nature"};
```

```

Random _rand = new Random();

for(int i=1; i<=10; i++){

    String sentence = sentences[_rand.nextInt(sentences.length)];

    String val = "ID " + i + " " + sentence;

    boolean queueSucess = kestrelClient.queue(queueName, val);

    System.out.println("queueSucess=" +queueSucess+ " [" + val +"]");
}
}

```

## 将项目从 **Kestrel** 移除

此方法将队列中的项目排队，而不是删除。` private static void dequeueItems(KestrelClient kestrelClient, String queueName) throws IOException, ParseError { for(int i=1; i<=12; i++){

```

Item item = kestrelClient.dequeue(queueName);

if(item==null){
    System.out.println("The queue (" + queueName + ") contains no items.");
}
else
{
    byte[] data = item._data;

    String receivedVal = new String(data);

    System.out.println("receivedItem=" + receivedVal);
}
}
```

此方法将队列中的项目排队，然后将其删除。

This method dequeues items from a queue and then removes them.

```

private static void dequeueAndRemoveItems(KestrelClient kestrelClient, String queueName)
throws IOException, ParseError
{
    for(int i=1; i<=12; i++){

        Item item = kestrelClient.dequeue(queueName);

        if(item==null){
            System.out.println("The queue (" + queueName + ") contains no items.");
        }
    }
}
```

```

        }
        else
        {
            int itemID = item._id;

            byte[] data = item._data;

            String receivedVal = new String(data);

            kestrelClient.ack(queueName, itemID);

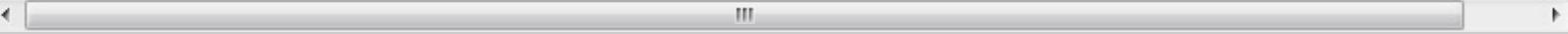
            System.out.println("receivedItem=" + receivedVal);
        }
    }
}

```

## ## 连续添加项目至 Kestrel

这是我们的最终运行程序，以便连续地将句子项添加到本地运行的Kestrel服务器的名为 **\*\*sentence\_queue\*\*** 的队列中。

为了阻止它在控制台中键入一个关闭括号 `char ']'`，然后按 'Enter'。



```

import java.io.IOException;
import java.io.InputStream;
import java.util.Random;

import org.apache.storm.spout.KestrelClient;
import org.apache.storm.spout.KestrelClient.Item;
import org.apache.storm.spout.KestrelClient.ParseError;

public class AddSentenceItemsToKestrel {

    /**
     * @param args
     */
    public static void main(String[] args) {

        InputStream is = System.in;

        char closing_bracket = ']';

        int val = closing_bracket;

        boolean aux = true;

        try {

            KestrelClient kestrelClient = null;
            String queueName = "sentence_queue";

            while(aux){

```

```

kestrelClient = new KestrelClient("localhost",22133);

queueSentenceItems(kestrelClient, queueName);

kestrelClient.close();

Thread.sleep(1000);

if(is.available()>0){
    if(val==is.read())
        aux=false;
}
}

} catch (IOException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}

catch (ParseError e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}

catch (InterruptedException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}

System.out.println("end");

}

}

```

## ## 使用 KestrelSpout

该拓扑结构使用KestrelSpout从Kestrel队列中读取句子，将句子分解成其组成词(Bolt: SplitSentence)，然后为每个

```

TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("sentences", new KestrelSpout("localhost",22133,"sentence_queue",new StringScheme(
builder.setBolt("split", new SplitSentence(), 10)
    .shuffleGrouping("sentences");
builder.setBolt("count", new WordCount(), 20)
    .fieldsGrouping("split", new Fields("word")));

```

## ## 执行

首先，在生产或开发模式下启动您本地的Kestrel服务器。

等大约5秒，以避免ConnectionException。

现在执行程序将项目添加到队列并启动Storm 拓扑。启动程序的排序并不重要。

如果您使用TOPOLOGY\_DEBUG运行拓扑，您应该会看到在拓扑中发出的元组。

# Storm 高级

# 针对 Storm 定义一个不是 JVM 的 DSL

针对 Storm 开始学习如何使用非 JVM 的 DSL 的正确的地址是 [storm-core/src/storm.thrift](#). 由于 Storm topologies 只是 Thrift 的结构, Nimbus 是 Thrift 守护进程, 您可以使用任何语言创建和提交 topologies.

当你针对 spouts 和 bolts 创建 Thrift 结构时, spout 和 bolt 的代码指定在 ComponentObject 结构中:

```
union ComponentObject {  
    1: binary serialized_java;  
    2: ShellComponent shell;  
    3: JavaObject java_object;  
}
```

针对 Python DSL, 您需要使用 "2". ShellComponent 可以让你指定一个脚本来运行组件 (例如, 你的 python 代码). 并且针对组件 (Storm 将使用反射来创建 spout 和 bolt) JavaObject 可以让你指定本地的 java spouts 和 bolts.

以下是一个 "storm shell" 命令, 它可以提交 topology. 用法如下:

```
storm shell resources/ python topology.py arg1 arg2
```

Storm Shell resources/ 下的东西包装到一个 jar 中, 将 jar 上传到 Nimbus, 并像下面这样调用你的 topology.py 脚本:

```
python topology.py arg1 arg2 {nimbus-host} {nimbus-port} {uploaded-jar-location}
```

然后您可以使用 Thrift API 连接到 Nimbus, 并提交 topology, 将 {uploaded-jar-location} 传递到 submitTopology 方法. 作为参考, 这里是 submitTopology 定义:

```
void submitTopology(1: string name, 2: string uploadedJarLocation, 3: string jsonConf, 4: StormTopo
```

最后, 在非 JVM DSL 中要做的一件重要的事情是, 可以轻松地在一个文件 (bolts, spouts 和 topology 的定义) 中定义整个 topology.

# 多语言协议

本页介绍了Storm 0.7.1中的多语言协议。0.7.1之前的版本使用了一个有些不同的协议，文档位于 [\[here\]\(Storm-multi-language-protocol-\(versions-0.7.0-and-below\).html\)](#).

## Storm 多语言协议

### Shell 组件

通过**ShellBolt**,**ShellSpout**和**ShellProcess**类实现对多语言的支持。这些类实现**IBolt**和**ISpout**接口以及执行脚本的协议或程序通过**Shell**使用Java的**ProcessBuilder**类。

### 包装**Shell**脚本

默认情况下，**ShellPorcess**假定您的代码打包在您的jar的**resources**子目录下的拓扑Jar内， 默认情况会更改当前的工作目录，该可执行线程是从Jar中提取的资源目录。一个Jar没有存储其中文件的权限。这包括允许**Shell**脚本由操作系统加载和运行的执行位。因此，在大多数示例中，脚本具有`python mybolt.py`的形式，因为**python**可执行文件已经在主管上，**mybolt**忆打包在**jar**的资源目录中。

如果你想打包更复杂的东西，像一个新版本的**python**本身，你需要改用**blob**这个存储和一个支持权限的`.tgz` 档案。

可以看这个文档 [Blob Store](#) 有更加详细的说明怎么运送**jar**的细节。

使用**ShellBolt/ShellSpout**与可执行文件+脚本一起发布在**blob store cache**中。

```
changeChildCWD(false);
```

在**ShellBolt/ShellSpout**的构造函数中。**shell**命令将相对于工作者的**cwd**。哪里的资源链接。

所以如果我发送**python**与一个名为`newPython`和一个**python ShellSpout**的符号链接我并发送到`shell_spout.py`，我会有如下写法

```
public MyShellSpout() {  
    super("./newPython/bin/python", "./shell_spout.py");  
    changeChildCWD(false);  
}
```

输出字段

输出字段是Thrift拓扑定义的一部分。这就意味着当您在java中的multing时，您需要创建一个扩展ShellBolt的bolt, 实现IRichBolt, 并声明declareOutputFields(类似于ShellSpout)中的字段。

您可以学习更多关于 [Concepts](#)

## 协议序言

一个简单的协议是通过STDIN和STDOUT来实现的执行脚本或程序。与该过程交换的所有数据为JSON格式，几乎可以支持任何语言。

## 包装你的东西

要在集群上运行Shell组件，那就是shelled的脚本必须在jar中提供的resources/目录中给master。

但是，在本地机器的开发或测试过程中，资源目录只需要在类路径中。

## 协议

Notes:

- 该协议的两端使用线读机制，所以一定要从输入中剪掉换行符并将其追加到输出中。
- 所有JSON输入和输出都由包含"end"的单行终止。请注意，此分隔符本身不是JSON编码的。
- 下面的项目符号是从脚本作者的角度编写的STDIN和STDOUT。

## 初始化握手

两种类型的shell组件的初始化握手是相同的：

- STDIN: 设置信息。这是一个具有Storm配置，PID目录和拓扑上下文的JSON对象，像这样：

```
{  
    "conf": {  
        "topology.message.timeout.secs": 3,  
        // etc  
    },  
    "pidDir": "...",  
    "context": {  
        "task->component": {  
            "1": "example-spout",  
            "2": "__acker",  
            "3": "example-bolt1",  
            "4": "example-bolt2"  
        }  
    }  
}
```

```
        },
        "taskid": 3,
        // Everything below this line is only available in Storm 0.10.0+
        "componentid": "example-bolt"
        "stream->target->grouping": {
            "default": {
                "example-bolt2": {
                    "type": "SHUFFLE"}}}},
        "streams": ["default"],
        "stream->outputfields": {"default": ["word"]},
        "source->stream->grouping": {
            "example-spout": {
                "default": {
                    "type": "FIELDS",
                    "fields": ["word"]
                }
            }
        }
    }
    "source->stream->fields": {
        "example-spout": {
            "default": ["word"]
        }
    }
}
```

您的脚本应该在此目录中创建一个以其**PID**命名的空文件。例如，**PID**为**1234**，因此在目录中创建名为**1234**的空文件。这个文件让主管知道**PID**，以便稍后关闭该过程。

从Storm 0.10.0起，Storm发送到shell组件的上下文一直是大大增强包括可用于JVM组件的拓扑上下文的所有方面。一个关键的补充是能够确定拓扑结构中的shell组件的源和目标（即输入和输出）`stream->target->grouping` and `source->stream->grouping`字典。在这些嵌套字典的最内层，分组被表示为一个最低限度具有`type`键的字典，但也可以有一个`fields`键，指定FIELDS分组中涉及哪些字段。

- **STDOUT**: 你的PID，在JSON对象中，像 `{"pid": 1234}`。shell组件将PID记录到其日志中。

接下来会发生什么取决于组件的类型：

## Spouts

Shell spouts 是同步的. 其余的发生在一段时间(true)循环:

- STDIN: 下一个, ack, 激活, 停用或失败命令。

"next" 相当于 `ISpout's` 的 `nextTuple`。看起来就像：

```
{"command": "next"}
```

"ack" 看起来像:

```
{"command": "ack", "id": "1231231"}
```

"activate" 相当于ISpout's的 `activate: {"command": "activate"}`

"deactivate" 相当于ISpout's的 `deactivate: {"command": "deactivate"}`

"fail" 看起来像:

```
{"command": "fail", "id": "1231231"}
```

- **STDOUT**: 您以前命令的输出结果。这可以是一系列发射和日志。

An emit looks like:

```
{  
  "command": "emit",  
  // The id for the tuple. Leave this out for an unreliable emit. The id can  
  // be a string or a number.  
  "id": "1231231",  
  // The id of the stream this tuple was emitted to. Leave this empty to emit to default stream.  
  "stream": "1",  
  // If doing an emit direct, indicate the task to send the tuple to  
  "task": 9,  
  // All the values in this tuple  
  "tuple": ["field1", 2, 3]  
}
```

如果不直接执行emit,则将立即收到STDIN上以元数组发布的元组为JSON数组。

"log" 将在工作日志中记录一条消息。看起来像:

```
{  
  "command": "log",  
  // the message to log  
  "msg": "hello world!"  
}
```

- **STDOUT**: "sync"命令结束发射和日志的顺序。看起来像:

```
{"command": "sync"}
```

同步之后，ShellSpout将不会读取您的输出，直到它发送另一个next,ack, 或fail命令。

请注意，与ISpout类似，工作人员的所有spouts将在下一次，确认或失败后被锁定，直到您同步。也像ISpout,如果没有元组为下一个发出，您应该睡眠少量的时间才能同步。  
ShellSpout不会自动为您做睡眠。

## Bolts

The shell bolt 是异步的. 您将在STDIN上收到元组，只要它们可用，您可以发出，确认或失败，并随时通过写入SDTOUT，如下所示:

- STDIN: 一个元组！这是一个这样的JSON编码结构:

```
{  
    // The tuple's id - this is a string to support languages lacking 64-bit precision  
    "id": "-6955786537413359385",  
    // The id of the component that created this tuple  
    "comp": "1",  
    // The id of the stream this tuple was emitted to  
    "stream": "1",  
    // The id of the task that created this tuple  
    "task": 9,  
    // All the values in this tuple  
    "tuple": ["snow white and the seven dwarfs", "field2", 3]  
}
```

- STDOUT: An ack, fail, emit, or log. Emits look like:

```
{  
    "command": "emit",  
    // The ids of the tuples this output tuples should be anchored to  
    "anchors": ["1231231", "-234234234"],  
    // The id of the stream this tuple was emitted to. Leave this empty to emit to default stream.  
    "stream": "1",  
    // If doing an emit direct, indicate the task to send the tuple to  
    "task": 9,  
    // All the values in this tuple  
    "tuple": ["field1", 2, 3]  
}
```

如果不直接执行emit，那么您将会收到在STDIN上发布元组的任务ids作为JSON数组。请注意，由于异步性质的shell bolt协议，当你读后你可以收不到任务的ids。你可以改为阅读要处理的先前发布或新元组的任务ids。但是你将按照相应的排放顺序接收任务id列表。

An ack 看起来像:

```
{  
    "command": "ack",  
    // the id of the tuple to ack
```

```
"id": "123123"  
}
```

A `fail` 看起来像:

```
{  
  "command": "fail",  
  // the id of the tuple to fail  
  "id": "123123"  
}
```

A `"log"` 将在工作日志中记录一条消息。看起来像:

```
{  
  "command": "log",  
  // the message to log  
  "msg": "hello world!"  
}
```

- 请注意，从0.7.1版本起，不再需要一个**shell bolt**进行‘同步’操作。

## 心跳处理 (0.9.3 及以上)

直到Storm 0.9.3, 心跳在**ShellSpout/ShellBolt**与它们之间多个子进程检测挂/子进程。任何通过多镜头与Storm进行连接的库，必须对听筒采取以下措施:

### Spout

**Shell spouts** 是同步的,因此子流程总是在`next()`的末尾发送`sync`命令，所以你不必为支持`spouts`的心跳做很多工作。也就是说，在`next()`期间，不要让子进程睡眠超过工作超时。

### Bolt

**Shell bolts** 是异步的，因此**ShellBolt**将定期向其子进程发送心跳元组。心跳元组看起来像:

```
{  
  "id": "-6955786537413359385",  
  "comp": "1",  
  "stream": "__heartbeat",  
  // this shell bolt's system task id  
  "task": -1,  
  "tuple": []  
}
```

当子进程接收到心跳元组时，它必须发送一个`sync`命令回到**ShellBolt**。

# Storm 内部实现

---

wiki 的这一部分专门解释 Storm 是如何实现的. 在阅读该部分之前, 您应该很好地掌握如何使用 Storm.

- [代码库的结构](#)
- [topology 的生命周期](#)
- [message 传递实现](#)
- [Metrics](#)
- [Nimbus HA](#)
- [Storm SQL](#)